Formal Methods

LNCS 14934

André Platzer
Kristin Yvonne Rozier
Matteo Pradella
Matteo Rossi (Eds.)

Formal Methods

26th International Symposium, FM 2024
Milan, Italy, September 9–13, 2024
Proceedings, Part II

2 Part II

Springer

OPEN ACCESS

Lecture Notes in Computer Science **14934**

Founding Editors

Gerhard Goos
Juris Hartmanis

Editorial Board Members

Elisa Bertino, USA
Wen Gao, China
Bernhard Steffen, Germany
Moti Yung, USA

Formal Methods

Subline of Lecture Notes in Computer Science

Subline Series Editors

Ana Cavalcanti, *University of York, UK*
Marie-Claude Gaudel, *Université de Paris-Sud, France*

Subline Advisory Board

Manfred Broy, *TU Munich, Germany*
Annabelle McIver, *Macquarie University, Sydney, NSW, Australia*
Peter Müller, *ETH Zurich, Switzerland*
Erik de Vink, *Eindhoven University of Technology, The Netherlands*
Pamela Zave, *AT&T Laboratories Research, Bedminster, NJ, USA*

More information about this series at https://link.springer.com/bookseries/558

Andre Platzer · Kristin Yvonne Rozier ·
Matteo Pradella · Matteo Rossi
Editors

Formal Methods

26th International Symposium, FM 2024
Milan, Italy, September 9–13, 2024
Proceedings, Part II

Springer

Editors
Andre Platzer
Karlsruhe Institute of Technology
Karlsruhe, Germany

Kristin Yvonne Rozier
Iowa State University
Ames, IA, USA

Matteo Pradella
Politecnico di Milano
Milan, Italy

Matteo Rossi
Politecnico di Milano
Milan, Italy

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-71176-3 ISBN 978-3-031-71177-0 (eBook)
https://doi.org/10.1007/978-3-031-71177-0

© The Editor(s) (if applicable) and The Author(s) 2025. This book is an open access publication.

Open Access This book is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.
The images or other third party material in this book are included in the book's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

If disposing of this product, please recycle the paper.

Preface

These volumes contain the papers presented for publication at the 26th International Symposium on Formal Methods (FM 2024), held in Milano, Italy, during September 9–13, 2024.

FM 2024 is the 26th event in the series of symposia organized by Formal Methods Europe (FME), an independent association whose aim is to stimulate the use of, and research on, formal methods for software development. The FM symposia have been successful in bringing together participants from academia, industry, and governments around a program of original papers on research and industrial experience, workshops, tutorials, reports on tools, projects, and ongoing doctoral research. FM 2024 is both an occasion to celebrate and a platform for enthusiastic researchers and practitioners from a diversity of backgrounds to exchange their ideas and share their experiences.

In addition to the main research track, FM 2024 included an Embedded Systems track, an Industry Day (I-Day) track, a Tutorial Paper track, a Journal First track, and a Doctoral Symposium. Also, 5 conferences and 6 workshops were co-located with FM 2024.

FM 2024 featured keynotes by David Basin (ETH Zürich), Hadas Kress-Gazit (Cornell University) and Marta Kwiatkowska (University of Oxford) with Byron Cook (University College London and AWS) as joint speaker for I-Day and the co-located conference on Formal Methods for Industrial Critical Systems (FMICS).

One main innovation of FM 2024 is the addition of a tutorial paper category. Tutorial papers present ideas with a focus on pedagogy over technical advances. By being written in a broadly-accessible way, tutorials clarify important ideas, bring new researchers into the community, and serve as a bridge to practitioners.

With 219 submissions, FM 2024 received a record-breaking number of paper submissions, which made it possible to select a particularly strong program. The main FM 2024 track received 178 submissions (143 regular research submissions, 8 case study submissions, 21 long tool paper submissions, 6 short tool demonstration submissions). The special embedded systems track of FM 2024 received 17 embedded submissions, the new tutorial paper track received 14 tutorial submissions, and the I-Day track received 10 industry report submissions. All paper submissions complying with the submission guidelines were reviewed by at least 3 reviewers, with a short author feedback period for a subset of the submissions selected for clarification and feedback by the 48 PC members. The main FM track accepted 44 papers (31 regular research papers, 1 case study paper, 8 long tool papers, 4 short tool demonstration papers) giving a 25% acceptance rate. The embedded systems track accepted 6 papers, the tutorial paper track accepted 10 papers, and the I-day track accepted 6 papers (3 regular papers, 2 case study papers, 1 extended abstract). Finally, 5 papers were selected for the Journal First track, and the Doctoral Symposium received 15 submissions (neither the journal first track papers nor the doctoral symposium ones appear in these proceedings).

FM 2024 invited the authors of all accepted papers to optionally submit an *artifact* —i.e., any additional material such as software, data sets, log files, machine-checkable proofs, etc., that substantiates the claims made in the paper—to the FM 2024 Artifact Evaluation Committee (AEC). After a short quick-check phase three AEC members reviewed each artifact in terms of consistency with and reproducibility of results presented in the paper, completeness, documentation and ease of (re-)use, and availability in an online repository with a DOI. Based on these reviews, and strictly following the EAPLS guidelines for artifact badging,[1] every artifact was awarded up to two badges:

Available. Artifacts that are publicly archived in a permanent way with a DOI that are in some way "relevant to" and "add value beyond the text in the article" are awarded the *available* badge.

Functional. Artifacts that are documented (containing at least an inventory and "sufficient description to enable the artifacts to be exercised"), consistent (i.e., "relevant to the associated paper, and significantly contribute to the generation of its main results"), complete ("as far as possible"), and exercisable, receive the *functional* badge.

Reusable. *Functional* and *available* artifacts that are "very carefully documented and well-structured to the extent that reuse and repurposing is facilitated" receive the *reusable* badge instead of the *functional* one.

Of the 45 submitted artifacts, 42 received the *available* badge, 18 were *functional*, and 14 were awarded the (*functional* and) *reusable* badge.

We are exceedingly grateful to everyone involved in making FM 2024 a success. We appreciate, in particular, the support by the FME board in all difficult decisions and are grateful to all PC members, Artifact Evaluation Commitee members, and subreviewers for volunteering their time in reviewing the submissions to FM, which was particularly challenging in light of the record high number of submissions, and for discussing papers thoroughly toward reaching consensus decisions. We also thank the other committees responsible for the Tutorial Paper track, Embedded Systems track, I-Day track, Journal First track, Doctoral Symposium, and workshops.

Finally we thank Springer for publishing these proceedings in the FM subline of LNCS and appreciate EasyChair in managing the paper submissions, reviewing, and proceedings compilation process.

July 2024

André Platzer
Kristin Yvonne Rozier
Matteo Pradella
Matteo Rossi

[1] https://eapls.org/pages/artifact_badges/eapls.org/pages/artifact_badges.

Organization

Program Committees

Research Track

André Platzer (Co-chair)	Karlsruhe Institute of Technology, Germany
Kristin Yvonne Rozier (Co-chair)	Iowa State University, USA
Erika Abraham	RWTH Aachen University, Germany
Wolfgang Ahrendt	Chalmers University of Technology, Sweden
Dalal Alrajeh	Imperial College London, UK
Luís Soares Barbosa	University of Minho, Portugal
Gilles Barthe	MPI-SP/IMDEA Software Institute, Spain
Dirk Beyer	LMU Munich, Germany
Pablo Castro	Universidad Nacional de Rio Cuarto, Argentina
Ana Cavalcanti	University of York, UK
Milan Ceska	Brno University of Technology, Czech Republic
Marsha Chechik	University of Toronto, Canada
Alessandro Cimatti	Fondazione Bruno Kessler, Italy
Alexandre Duret-Lutz	EPITA Research Laboratory (LRE), France
Marie Farrell	University of Manchester, UK
Orna Grumberg	Technion - Israel Institute of Technology, Israel
Arie Gurfinkel	University of Waterloo, Canada
Anne E. Haxthausen	Technical University of Denmark, Denmark
Marieke Huisman	University of Twente, The Netherlands
Reiner Hähnle	TU Darmstadt, Germany
Peter Höfner	Australian National University, Australia
Einar Broch Johnsen	University of Oslo, Norway
Joost-Pieter Katoen	RWTH Aachen University, Germany
Nikolai Kosmatov	Thales Research & Technology, France
Orna Kupferman	Hebrew University, Israel
Peter Lammich	University of Twente, The Netherlands
Martin Leucker	University of Lübeck, Germany
Jianwen Li	East China Normal University, China
Ravi Mangal	Colorado State University, USA
Mieke Massink	CNR, Italy
Anastasia Mavridou	KBR/NASA, USA
Annabelle McIver	Macquarie University, Australia
Claudio Menghi	University of Bergamo, Italy
Stefan Mitsch	DePaul University, USA
Cesar Munoz	NASA, USA
Aniello Murano	University of Naples Federico II, Italy

Violet Ka I Pun	Western Norway University of Applied Sciences, Norway
Zvonimir Rakamaric	Amazon Web Services, USA
Philipp Rümmer	University of Regensburg, Germany
Cristina Seceleanu	Mälardalen University, Sweden
Natasha Sharygina	Università della Svizzera italiana, Switzerland
Jun Sun	Singapore Management University, Singapore
Lucas Martinelli Tabajara	Rice University, USA
Yong Kiam Tan	A*STAR, Singapore
Stefano Tonetta	Fondazione Bruno Kessler, Italy
Georg Weissenbacher	TU Wien, Austria
Virginie Wiels	ONERA/DTIS, France
Huan Xu	University of Maryland, USA
Naijun Zhan	Chinese Academy of Sciences, China
Shufang Zhu	University of Oxford, UK

Embedded Systems Track

Alessandro Cimatti (Chair)	Fondazione Bruno Kessler, Italy
Frédéric Boulanger	CentraleSupélec, France
Lei Bu	Nanjing University, China
Qinxiang Cao	Shanghai Jiao Tong University, China
Liqian Chen	National University of Defense Technology, China
Martin Fränzle	Carl von Ossietzky Universität Oldenburg, Germany
Paula Herber	University of Münster, Germany
Inigo Incer	California Institute of Technology, USA
Ahmed Irfan	SRI International, USA
Eunsuk Kang	Carnegie Mellon University, USA
Sergio Mover	École Polytechnique, France
Dejan Nickovic	AIT Austrian Institute of Technology, Austria
Pierluigi Nuzzo	University of Southern California, USA
Roberto Passerone	University of Trento, Italy
Heyuan Shi	Central South University, China
Fu Song	Chinese Academy of Sciences, China
Cong Tian	Xidian University, China
Stavros Tripakis	Northeastern University, USA

Tutorial Papers Track

Shriram Krishnamurthi (Co-chair)	Brown University, USA
Luigia Petre (Co-chair)	Åbo Akademi University, Finland
Anindya Banerjee	IMDEA Software Institute, Spain
Nikolaj Bjørner	Microsoft, USA
Marcello Bonsangue	Leiden University, The Netherlands
David Thrane Christiansen	Lean FRO, LLC, Denmark
Brijesh Dongol	University of Surrey, UK

Jan Friso Groote	TU Eindhoven, The Netherlands
Stefan Hallerstede	Aarhus University, Denmark
Daniel Jackson	Massachusetts Institute of Technology, USA
Jeroen Keiren	TU Eindhoven, The Netherlands
Markus Alexander Kuppe	Microsoft, USA
Thierry Lecomte	CLEARSY, France
Jannis Limperg	LMU Munich, Germany
Rosemary Monahan	Maynooth University, Ireland
Tim Nelson	Brown University, USA
Maurice ter Beek	CNR, Italy

Industry Day Track

Oksana Tkachuk (Co-chair)	Amazon Web Services, USA
Tim Willemse (Co-chair)	TU Eindhoven, The Netherlands
Nikolaj Bjørner	Microsoft, USA
Jennifer Davis	Collins Aerospace, USA
Leo Freitas	Newcastle University, UK
Dimitra Giannakopoulou	Amazon Web Services, USA
Mario Gleirscher	University of Bremen, Germany
Cláudio Gomes	Aarhus University, Denmark
Klaus Havelund	California Institute of Technology, USA
Nikolai Kosmatov	Thales Research & Technology, France

Artifact Evaluation

Carlos E. Budde (Co-chair)	Università di Trento, Italy
Arnd Hartmanns (Co-chair)	University of Twente, The Netherlands
Jie An	Chinese Academy of Sciences (ISCAS), China
Alberto Bombardelli	Fondazione Bruno Kessler, Italy
Konstantin Britikov	Università della Svizzera italiana, Switzerland
Laura Bussi	CNR, Italy
Julie Cailler	LIRMM, France
Emily Clement	Université Paris Cité, CNRS, IRIF, France
César Cornejo	Universidad Nacional de Rio Cuarto, Argentina
Yanni Dong	University of Twente, The Netherlands
Daniel Drodt	TU Darmstadt, Germany
Federico Formica	McMaster University, Canada
Fabrizio Fornari	University of Camerino, Italy
Laura P. Gamboa Guzman	Iowa State University, USA
Rong Gu	Mälardalen University, Sweden
Long H. Pham	Singapore Management University, Singapore
Tobias John	University of Oslo, Norway
Aditi Kabra	Carnegie Mellon University, USA
Mehrdad Karrabi	Institute of Science and Technology Austria, Austria
Paul Kobialka	University of Oslo, Norway
Marian Lingsch-Rosenfeld	LMU Munich, Germany

Alexander Mackay	Australian National University, Australia
Andrea Manini	Politecnico di Milano, Italy
Antoine Martin	EPITA Research Laboratory (LRE), France
Lucas Martinelli Tabajara	Rice University, USA
Tobias Nießen	TU Wien, Austria
Tommaso Oss	University of Trento, Italy
Quentin Peyras	ONERA, France
Andrea Pferscher	University of Oslo, Norway
Roberto Pizziol	IMT School for Advanced Studies Lucca, Italy
Francesco Pontiggia	TU Wien, Austria
Edoardo Putti	University of Twente, The Netherlands
Florian Renkin	Université Paris Cité, IRIF, France
Guillermo Román-Díez	Universidad Politécnica de Madrid, Spain
Alec Rosentrater	Iowa State University, USA
Lorenzo Rossi	University of Camerino, Italy
Ömer Sayilir	University of Twente, The Netherlands
Philipp Schlehuber-Caissier	EPITA Research Laboratory (LRE), France
Riccardo Sieve	University of Oslo, Norway
Reza Soltani	University of Twente, The Netherlands
Alexander Stekelenburg	University of Twente, The Netherlands
Jack Stodart	Australian National University, Australia
Emily Yu	Institute of Science and Technology Austria, Austria

Journal First Track

Michael Butler (Chair)	University of Southampton, UK
Dines Bjørner	Technical University of Denmark, Denmark
Eerke Boiten	De Montfort University, UK
Maurice ter Beek	CNR, Italy

Doctoral Symposium

Carlo A. Furia (Co-chair)	Università della Svizzera italiana, Switzerland
Laura Kovács (Co-chair)	TU Wien, Austria
Wolfgang Ahrendt	Chalmers University of Technology, Sweden
Marcello M. Bersani	Politecnico di Milano, Italy
Nikolaj Bjørner	Microsoft, USA
Paula Herber	University of Münster, Germany
Marieke Huisman	University of Twente, The Netherlands
Alexandra Mendes	University of Porto, Portugal
Rosemary Monahan	Maynooth University, Ireland
Raúl Pardo	IT University of Copenhagen, Denmark
Simon Robillard	Université de Montpellier, France
Silvia Lizeth Tapia Tarifa	University of Oslo, Norway
Stefano Tonetta	Fondazione Bruno Kessler, Italy
Mattias Ulbrich	Karlsruhe Institute of Technology, Germany

FME Board

Ana Cavalcanti	University of York, UK
Maurice ter Beek	CNR, Italy
Nico Plat	Thanos, The Netherlands
Lars-Henrik Eriksson	Uppsala University, Sweden
Einar Broch Johnsen	University of Oslo, Norway

Organization Committee

General Chairs

Matteo Pradella	Politecnico di Milano, Italy
Matteo Rossi	Politecnico di Milano, Italy

Sponsorship and Exhibition Chairs

Marcello M. Bersani	Politecnico di Milano, Italy
Michele Chiari	TU Wien, Austria

Social Media Chair

Livia Lestingi	Politecnico di Milano, Italy

Workshop Chairs

Stefania Gnesi	CNR, Italy
Marieke Huisman	University of Twente, The Netherlands

Additional Reviewers

Yehia Abd Alrahman
Emma Ahrens
Aliyu Tanko Ali
Shaull Almagor
José Bacelar Almeida
Roman Andriushchenko
Santiago Arranz-Olmos
Anagha Athavale
Ziggy Attala
Giorgio Audrito
Peter Backeman
Daniel Baier
Jialu Bao
Chinmayi Prabhu Baramashetru
Davide Basile
Ludovico Battista
Kevin Batz
Anna Becchi
Valeria Bengolea
Raphaël Berthon
Lionel Blatter
Martin Blicha
Alberto Bombardelli
Frédéric Boniol
Alexander Bork
Konstantin Britikov
Christopher Brix
Julien Brunel
Richard Bubel
Julie Cailler
Georgiana Caltais
Mishel Carelli
Valentin Cassano
Valentina Castiglioni
Davide Catta
Claudia Cauli

David Chemouil
Mingshuai Chen
Xin Chen
Felix Cherubini
Po-Chun Chien
Vincenzo Ciancia
Davide Davoli
André De Matos Pedro
Erik De Vink
Ramiro Demasi
Daniel Drodt
Manuel Eberl
Zafer Esen
Grigory Fedyukovich
Marco A. Feliu
Nick Feng
Shenghua Feng
Anthony Fernandes Pires
Angelo Ferrando
Carla Ferreira
Joao F. Ferreira
Ira Fesefeldt
Paul Fiterau-Brostean
Simon Foster
Luis Garcia
Christina Gehnen
Tiberiu A. Georgescu
Marcus Gerhold
Roland Glück
Michał Tomasz Godziszewski
R. Govind
Srajan Goyal
Lukas Graussam
Alberto Griggio
Lukas Grätz
Rong Gu
Vojtěch Havlena
Holly Hendry
Paula Herber
Roland Herrmann
Hans-Dieter Hiep
Raik Hipler
Sebastian Holler
Lukáš Holík
Jacob Howe
Aditi Kabra
Hannes Kallwies
Eduard Kamburjan
Emin Karayel
Jeroen J. A. Keiren
Ata Keskin
Matthias Kettl
Karam Kharraz
Bram Kohlen
Tomáš Kolárik
Katherine Kosaian
József Kovács
Gereon Kremer
Harald König
Faezeh Labbaf
Martin Lange
Jonathan Laurent
Tristan Le Gall
Nham Le
Thomas Lemberger
Ondrej Lengal
Yong Li
Chencheng Liang
Marian Lingsch-Rosenfeld
Debasmita Lohar
Delphine Longuet
Michele Loreti
Andreas Lööw
Filip Macák
Alexandre Madeira
Vadim Malvone
Lina Marsso
Manuel A. Martins
Alexandra Mendes
Robert Mensing
Hannah Mertens
Munyque Mittelmann
Alvaro Miyazawa
Mariano Moscato
Mohammadreza Mousavi
Sergio Mover
Logan Murphy
Muhammad Naeem
Jasper Nalbach
Renato Neves
Kim Nguyen
Thomas Noll

Jose Oliveira
Rodrigo Otoni
Gianmarco Parretti
Mário Pereira
Quentin Peyras
Adrien Pommellet
Siddharth Priya
José Proença
Valentin Promies
Edoardo Putti
Tim Quatmann
Willard Rafnsson
Itsaka Rakotonirina
Omer Rappoport
António Ravara
Gianluca Redondi
Germán Regis
Andrew Reynolds
Pedro Ribeiro
Martin Sachenbacher
Augusto Sampaio
Abhiroop Sarkar
Jonas Schiffl
Philipp Schlehuber-Caissier
Philipp Schröer
Roberto Sebastiani
Filipo Sharevski
Xujie Si
Teofil Sidoruk
Julien Signoles
Joseph Slagel
Jorge Sousa Pinto
Francesco Spegni
Daniel Stan
Martin Steffen
Alexander Stekelenburg
Volker Stolz
Han Su
Roger Su
Yusen Su
Silvia Lizeth Tapia Tarifa
Philip Tasche
Samuel Teuber
Daniel Thoma
Chun Tian
Gan Ting
Laura Titolo
Noriko Tomuro
Dmitriy Traytel
Mattias Ulbrich
Tom van Dijk
Andrea Vandin
Mahsa Varshosaz
Hari Govind Vediramana Krishnan
Franck Vedrine
Adele Veschetti
Henrik Wachowitz
Philipp Wendler
Hao Wu
Yechuan Xia
Shengping Xiao
Norihiro Yamada
Fang Yan
Tengshun Yang
Kangfeng Ye
Lina Ye
Bohua Zhan
Zhi Zhang
Hengjun Zhao
Xingyu Zhao
Ghiles Ziat
Martin Zimmermann
Paolo Zuliani

Contents – Part II

Contents – Part I

Tools and Case Studies

Extending Isabelle/HOL's Code Generator with Support for the Go Programming Language

Terru Stübinger[1,2] and Lars Hupel[1,2](✉)

[1] Giesecke+Devrient, Prinzregentenstr. 161, 81677 München, Germany
[2] Technische Universität München, School of Computation, Information and Technology, Boltzmannstr. 3, 85748 Garching bei München, Germany
stuebinm@in.tum.de, lars.hupel@tum.de

Abstract. The Isabelle proof assistant includes a small functional language, which allows users to write and reason about programs. So far, these programs could be extracted into a number of functional languages: Standard ML, OCaml, Scala, and Haskell. This work adds support for Go as a fifth target language for the Code Generator. Unlike the previous targets, Go is not a functional language and encourages code in an imperative style, thus many of the features of Isabelle's language (particularly data types, pattern matching, and type classes) have to be emulated using imperative language constructs in Go. The developed Code Generation is provided as an add-on library that can be simply imported into existing theories.

Keywords: Theorem provers · Code generation · Go programming language

1 Introduction

The interactive theorem prover *Isabelle* of the LCF tradition [13] is based on a small, well-established and trusted mathematical inference kernel written in Standard ML. All higher-level tools and proofs, such as those included in the most commonly-used logic *Isabelle/HOL*, have to work through this kernel.

Many of the tools available to users in *Isabelle/HOL* feel immediately familiar to anyone with experience in functional programming languages: it is possible to define data types, functions, and Haskell-style type classes and instances.

Isabelle's nature as a theorem prover further makes it easy to formalise and prove propositions about such programs. To allow use of such programs outside of the proof assistant's environment, Isabelle comes equipped with a *Code Generator*, allowing users to extract source code in Haskell, Standard ML, Scala, or OCaml, which can then be compiled and executed. This translation of code works by first translating into an intermediate language called *Thingol*, shared between all targets; from this language, code is then transformed into the individual target languages via the principle of *shallow embedding*, that is, by representing

© The Author(s) 2025

constructs of the source language using only a well-defined subset of the target language, thus side-stepping the issue of finding a complete formal description of a target language's behaviour [6,7].

Go is a programming language introduced by Google in 2009. It is a general-purpose, garbage-collected, and statically typed language [4]. In contrast to the existing targets of Isabelle's Code Generator, it is not a functional language, and encourages programming in an imperative style. However, it is a very popular language, and many large existing code bases have been written in it.

Contributions. This paper extends Isabelle's Code Generation facility with support for Go. For that, we demonstrate a translation scheme from programs in Thingol to programs in Go (§4). We provide this facility as a stand-alone theory file that can easily be imported into existing developments. We provide our development as an entry in the *Archive of Formal Proofs (AFP)*—a repository of Isabelle proof libraries—, making it immediately usable in other contexts [17].

The motivation for this work stems from the internal use of both ecosystems at Giesecke+Devrient: Isabelle for formalisation purposes, and Go for the real-world implementation. This naturally lead to a formalisation gap, which this project sought to close (§5).

Related Work. This paper describes the first attempt at translating Isabelle formalisations into a non-functional programming language. Prior work in leveraging imperative features in the Code Generator [2] has targeted the existing, functional programming languages, and thereby could reuse much of the existing infrastructure. There is also unpublished work on adding support for F# to the Code Generator [1], another functional language.

Shallow embeddings of C in proof assistant are already well known; for example in F* [14], Isabelle [16], and Why3 [15]. Those tools are not designed to export arbitrary code, but require developers to use a restricted subset of the host language. Instead, they are mainly geared towards low-level programming; with some providing C-style memory management. Our work focuses instead on translating the full functional host language into a high-level imperative language, therefore avoiding the need to (re)write host language code specifically for the purpose.

2 The Intermediate Language Thingol

Isabelle's Code Generation pipeline works in multiple stages. Crucially, all definitions made in Isabelle are first translated into an abstract intermediate language called *Thingol*, which is the last step shared between all target languages. The final stage then uses a shallow embedding to translate the Thingol program into source code of the target language.

Consequently, Thingol's design reflects the features common to previous target languages, and is based on a simply-typed λ-calculus with ML-style polymorphism. Perhaps surprisingly, Thingol also supports type classes, which can be

```
datatype Nat = Zero | Suc Nat
datatype α list = Nil | Cons α (α list)

fun hd2 :: ∀α.α list ⇒ α option where
  hd2 xs = case xs of Nil ⇒ None
                    | Cons x Nil ⇒ None
                    | Cons x (Cons y xs) ⇒ Some y

class semigroup where
  (+) :: α ⇒ α ⇒ α

class monoid ⊆ semigroup where
  zero :: α

instance Nat :: semigroup where
  a + Zero = a
  Zero + a = a
  (Suc a) + b = Suc (a + b)

instance Nat :: monoid where
  zero = Zero

fun sum :: (α :: monoid) list ⇒ α where
  sum xs = fold (+) xs zero
```

Fig. 1. An example program (omitting the definition of `fold` for brevity)

mapped easily to Haskell and Scala, but less easily to the other targets, which instead use a dictionary construction (§4.5). The supported fragment of type classes and instance corresponds to Haskell98, with the exception of constructor classes (which would require a more expressive type system) [8,10].

Thingol's terms are simple λ-expressions with the addition of case expression for pattern matching on data types. A Thingol program is a list of declarations, i.e. top-level items which introduce data types, functions, type classes, and their instances.

While there is no formal semantics of Thingol, it can be thought of as a *Higher-Order Rewrite System* (HRS) [11,12]. It provides a convenient abstraction over the target languages' semantics. Because a HRS does not have a specified evaluation order, the Code Generator cannot guarantee total, but only partial correctness. (This restriction applies to all supported target languages.)

Reusing Thingol has two immediate benefits: we can leverage the entire entire existing pipeline as well as its existing code adaptations, and are not forced to reimplement some tedious translation of Isabelle's more advanced features. Additionally, creating a custom intermediate language would not help to meaningfully address the functional–imperative mismatch between Isabelle/HOL and Go, but only shift the complexity elsewhere.

3 The Target Fragment of Go

Go, being an imperative language, differs in many aspects from the already-existing target languages of Isabelle's Code Generator. Conversely, many of Go's unique features are not needed by the generator. Since the translation works as a shallow embedding into the target language, it suffices to use the fragment which can be used to represent the various statements of Thingol. Consequently, we will focus on this fragment only, but discuss—if necessary—why we did not pursue alternative features or solutions.

This approach leaves many of Go's most interesting features (e.g. channels or methods) entirely unused. The fragment we use can be understood as a "functional subset" of the Go language, meaning that it comprises only those features that closely align with those of the (functional) pre-existing code generation targets available in Isabelle as well as those of Thingol.

3.1 Syntax

The syntactic fragment used by the Code Generator[1] is inspired by that of Featherweight Generic Go [5], but differs in some important aspects:

1. Methods are not included; instead we use "ordinary" top-level functions.
2. Go distinguishes syntactically between expressions and statements, whereas Featherweight Generic Go does not. We retain this distinction and discuss conversion between them in §3.4.
3. Type parameters can be declared with an interface constraint. However, in our fragment the only available constraint is the unconstrained `any`, as Go's other constraints are not useful for our translation (§4.5).
4. We use modern Go's syntax for generics, which differs from the one used by Featherweight Generic Go, which pre-dates the introduction of generics in Go 1.18 and was meant as a proposal demonstrating a possible design.

3.2 Declarations

A (top-level) declaration D can define either a new type or function. Within one package, the order of declarations does not matter; any item may reference any other. A program as a whole is simply a list $\overline{D}$ of such declarations (note that we use overlines such as $\overline{\alpha}$ to mean syntactic repetition).

Structure Types. A declaration of the form $\texttt{type}\ t_S\left[\overline{\alpha\ c}\right]\ \texttt{struct}\{\overline{A\ \tau}\}$ introduces a new type constructor with fields $\overline{A}$ of types $\overline{\tau}$ to the program. It may be polymorphic and take type arguments $\overline{\alpha}$ which can be freely referenced within $\overline{\tau}$. Since Go's syntax demands a constraint c for each type variable α, we always use `any`, which allows any type to be substituted for α.

Note that there is no analogous construct to Thingol's sum types; that is, it is not possible to a have a structure type which has more than one constructor.

[1] See the appendix of this paper for the full listing: arxiv.org/abs/2310.02704.

Therefore, when encountering non-trivial sum types in Thingol, we must encode them accordingly (see §4.2 for details).

Interface Types. A declaration of the form `type` $t_I\left[\overline{\alpha\ c}\right]$ `interface{}` introduces a new (empty) interface type to the program. While Go supports non-empty interfaces containing methods, we do not use this feature (see §4.5).

Unlike interfaces in typical object-oriented languages such as Java, Go's interfaces are structural in nature: any `struct` value conforms to an interface if (and only if) the `struct` implements a superset of the declared methods of the interface. This can also be probed at runtime.

This implies that empty interfaces correspond to a "top" type that can denote arbitrary values. Go defines the unconstrained interface `any` as an alias to this empty interface type, which we use extensively in the translation scheme of data types, for reasons that will be explained later (§4.2). Additionally, we also use them for the translation of type classes (§4.5).

Functions. A declaration `func` $f\left[\overline{\alpha\ c}\right]$`(`$\overline{x\ \tau}$`)` `(`$\overline{\gamma}$`)` `{` s `}` introduces a new function f to the program. The type parameters $\overline{\alpha}$ can be referenced within both argument types $\overline{\tau}$ and the return types $\overline{\gamma}$.

Unlike in Thingol, a function cannot have multiple equations nor perform pattern matching on its arguments. Instead there is only one list of argument names $\overline{\alpha}$, which are in scope for the (unique) function body s.

An unusual feature of Go is that its functions may return more than one value (note that we have return types $\overline{\gamma}$ instead of just a single return type γ):

```
func foo() (bool, int, string) {
  return false, 42, "bar"
}

func main() {
  x, y, z := foo()
}
```

At first glance this might seem analogous the tuples present in Standard ML, with `foo()` returning a single value of the tuple `(bool, int, string)`. But this is not the case; Go has no concept of tuples. Instead, the function itself returns multiple values, which must be immediately assigned names (or discarded) at the function's call site. Thus a call like `no_tuples := foo()` is not allowed.

3.3 Expressions

Expressions e can have several forms: variables, function application, and function abstraction are familiar from the λ-calculus. The others may require a bit more explanation.

Structure Literal. A literal of the form $t_s[\overline{\alpha}]\{\overline{e}\}$ gives a value of the `struct` type with name t_S applied to type arguments $\overline{\alpha}$, i.e., it produces a new value of the type $t_s\left[\overline{\alpha}\right]$ in which the fields are set to the evaluated forms of the expressions $\overline{e}$. Note that the field names present in the declaration of a `struct` type are absent: while they could be used, Go does not require them. We omit them in the interest of shorter code.

Field Selection. An expression $e.A$ selects the field named A of an expression e, which must have a fitting `struct` type τ_S that was declared with a field name A, and returns the value of that field. This is the only place outside a structure type's declaration that field names are used.

Type Conversion. An expression $\tau_I(e)$ evaluates to a value of the interface type τ_I which contains the evaluated form of e as its inner value. The original type σ of e is not erased at runtime; it can be recovered using a type assertion statement (see the next section). This expression can also be thought of as an "upcast".

3.4 Statements

Unlike in Isabelle (and in Thingol) where "everything is an expression", Go has the same syntactic distinction between expression and statements that is common across imperative languages: an assignment `x := 42;` is a statement, not an expression, and cannot be used in places where an expression is expected.

However, we constrain our fragment to only include sequences of statements that end in a `return`. This enables easy embedding of a statement into an expression: wrapping it into an immediately-called lambda `func ()` τ `{` s `}()` suffices. Note that a sequence of statements interspersed with `;` is treated syntactically as a single statement.

The remainder of this section introduces the statement forms of our fragment. All but the type assertion should feel familiar from similar languages.

Return. A statement `return`$\overline{e}$ evaluates one or more expressions, then returns from the current function. The $\overline{e}$ must match the return types given in the function's head.

If Statement. A statement of the form `if (`e`) {` s_1 `};` s_2 will evaluate e, which must have a boolean type. If it evaluates to the built-in value `true`, then s_1 is evaluated. Since all statements end in `return`, it will then return from the current function. Otherwise, s_2 is evaluated. The form `if (`e`) {` s `} else {` s_2 `}` would be semantically equivalent within our fragment; we avoid it to reduce nesting in the generated code.

Type Assertion. A statement of the form x`,` y `:=` e`.(`σ`)` can be thought of as the inverse operation of type conversions, i.e., a "downcast". For an expression e of an interface type τ_I, the assertion checks whether the inner value contained within the interface value has type σ. The boolean variable y will indicate if the check was successful. If so, x will be bound to that inner value; otherwise, it will be `nil`, Go's null pointer. Note that the type of x is σ.

4 Translation Scheme

In this section, we will discuss the concrete translation schemes employed for Thingol programs. For brevity, we omit purely syntactic mappings, and focus on the non-trivial steps.

The translation scheme attempts to preserve names as far as possible. Isabelle's Code Generator already provides (re)naming infrastructure, such as generating guaranteed-unused "fresh" names where necessary. In addition to that, some functions and data types require upper-case names, to match Go's rules for exported symbols.

4.1 Types, Terms and Statements

We define three translations $\textsc{type}(\tau)$, $\textsc{expr}(t)$, and $\textsc{stmt}(t)$. The first is a straightforward syntactic mapping of types. In the remainder of the chapter, we will informally equate Thingol types τ with their Go translation $\textsc{type}(\tau)$ and write both simply as τ. For now, we exclude any mapping of common types (e.g. integers) to built-in Go types; we will revisit this topic later (§4.6).

The other two translations—EXPR and STMT—are used for converting Thingol terms into Go expressions and statements. Which one is used thus depends on what Go expects in each particular context; for example, terms used as function arguments use EXPR; a term which is a function body uses STMT. Semantically, for any term t, EXPR and STMT satisfy the following equivalences:

$$\begin{aligned}\textsc{stmt}(t) &\equiv \texttt{return}\ \textsc{expr}(t)\texttt{;}\\ \textsc{expr}(t) &\equiv \texttt{func()}\ \tau\ \texttt{\{}\textsc{stmt}(t)\texttt{\}()}\end{aligned}$$

Abstractions. The translation of a λ-abstraction $\lambda(x{::}\tau).\ (t{::}\gamma)$ demonstrates the distinction between expressions and statements:

$$\textsc{expr}(\lambda(x{::}\tau).\ (t{::}\gamma)) = \texttt{func (}x\ \tau\texttt{)}\ \gamma\ \texttt{\{}\textsc{stmt}(t)\texttt{\}}$$

Although curried abstractions are unusual in Go, no effort is made to uncurry them (unlike top-level functions, which we do uncurry §4.4).

Applications of Top-Level Functions. Applications t are more tedious: Definitions of top-level functions are uncurried (§4.4), so we first have to check if t is a call to such a function, i.e., if t has the shape $\big(\cdots((f[\overline{\tau}_i]\ a_1)\ a_2)\cdots\big)\ a_n$, where f references a top-level function or data type constructor taking m arguments.

If so, we have to consider three cases:

1. Fully-satured application ($n = m$); all arguments are passed into f
2. Unsatured application ($n < m$); need to η-expand

3. Over-satured application ($n > m$). This occurs if f returns another function, with a_1 to a_m being the immediate arguments to f and any remaining a_{m+1} to a_n as curried arguments. The latter will be passed individually.

As will be described later (§4.5), the dictionary construction used to encode Isabelle's type classes may introduce additional (value-level) parameters to top-level functions, also adding corresponding additional arguments d_1 to d_r to each of their applications. These are inserted before the user-defined parameters.

Altogether, we arrive at the following scheme when f references a function:

$$\textsc{expr}(t) = f\,\texttt{[}\tau_1\texttt{,}\ldots\texttt{,}\tau_i\texttt{]}\,\texttt{(}d_1\texttt{,}\ldots d_r\texttt{,}a_1\texttt{,}\ldots\texttt{,}a_m\texttt{)}\,\texttt{(}a_{m+1}\texttt{)}\ldots\texttt{(}a_n\texttt{)}$$

Finally, if f references a data type constructor of a type τ rather than a function, the case $n > m$ cannot occur. However, we must wrap the constructor into a type conversion to type τ, and use slightly different syntax for passing the arguments:

$$\textsc{expr}(t) = \kappa\texttt{(}f\,\texttt{[}\tau_1\texttt{,}\ldots\texttt{,}\tau_i\texttt{]}\texttt{\{}d_1\texttt{,}\ldots d_r\texttt{,}a_1\texttt{,}\ldots\texttt{,}a_m\texttt{\}}\texttt{)}$$

Lambda Applications. If an application $t = t_1\ t_2$ is not a call to a top-level function, then the translation is straightforward: $\textsc{expr}(t_1\ t_2) = \textsc{expr}(t_1)\texttt{(}\textsc{expr}(t_2)\texttt{)}$.

4.2 Data Types

A data type κ defined in Thingol consists of type parameters $\overline{\alpha}_i$ and constructors $\overline{f}_i$. Each f_i gets translated into its own separate `struct` type.

As was discussed in §3, Go knows no sum types, thus the translation has to simulate their behaviour by other means. For a data type, we generate a new unconstrained interface type δ, meant to represent any constructor f_i of κ.

If the data type κ has exactly one constructor f_1, then no additional interface type δ is generated.

Constructors. Defining a `struct` type for an individual constructor is straightforward. A constructor f with fields of types τ_1 to τ_i is translated into Go as a `struct` with the same name and fields: `type` f `struct {`$\overline{A\ \tau_i}$`}`, where the $\overline{A}_i$ are newly-invented names for each of the fields, as no field names are present in Thingol. Note that those generated field names are entirely unimportant (access happens only through destructors, and the names are not required when constructing a value); the only requirement imposed on them is that each $\overline{A}_i$ of the same `struct` are distinct. Thus the type `Nat` (Fig. 1) becomes:

```
type Nat any;
type Zero struct { };
type Suc struct { A Nat; };
```

With that, we can construct the number 1 as `Nat(Suc{Nat(Zero{})})` . The interface type δ (here `Nat`) acts as a faux sum type: the translation promises that (as long as its input program was type-correct) it will never contain anything but values of types `Zero` and `Suc`. On the Go side, there is no such guarantee: it sees `Nat` as unconstrained, and would happily allow such values as `Nat(Suc{nil})` or even `Suc{"wrong"}`, leading to runtime exceptions elsewhere in the generated code, especially in translated pattern matches (§4.3).

Destructors. Along with each constructor's `struct` type, we generate a synthetic function f`_dest` not present in the Thingol program, to be used as a destructor in the translation of Thingol's case expressions (§4.3). Their sole purpose is to unpack and return the individual fields in a `struct` type, exploiting Go's multiple return types.

```
func f_dest (p f) (τ1, ..., τn) {
  return p.A1, ..., p.An
}
```

Destructors are omitted when there are no fields to unpack. For `Nat`, we need only one:

```
func Suc_dest(p Suc)(Nat) { return p.A; }
```

Example. Slightly more involved is the α`list` data type (Fig. 1). It is polymorphic, and thus requires use of Go's generics:

```
type List[a any] interface {};
type Nil[a any] struct { };
type Cons[a any] struct { A a; Aa List[a]; };
func Cons_dest[a any](p Cons[a])(a, List[a]) { return p.A, p.Aa; }
```

4.3 Case Expressions

Thingol's case expressions implement pattern matching on a value, in a way which will be immediately familiar from other functional languages such as Standard ML or Haskell: they inspect a *scrutinee* t and match it against a series of clauses $\overline{p_i \rightarrow b_i}$. Each clause contains a pattern p_i and a term t_i that is to be evaluated if the pattern matches the scrutinee. Syntactically, patterns are a subset of terms; they can only be composed of variables and fully-satisfied applications of data type constructors to sub-patterns $f\ \overline{p}_i$ constructed of the same subset.

Since Go has no comparable feature, a data type pattern in a case expression is translated into a series of (possibly nested) `if`-conditions and calls to destructor functions. The bodies of the innermost `if`-condition then correspond to the translated terms t_i, which must be in statement-form, i.e., ending in a `return`-statement. Thus, if the pattern could be matched, further patterns will not be executed. Naturally, using `return` in this manner implies that a case expression must always either be in tail position, or else be wrapped into an anonymous function if it does not (§3).

If the pattern did not match, execution will continue with either the next block of `if`-conditions generated from the next clause, or encounter a final catch-all call to Go's built-in `panic` function, which aborts the program in case of an incomplete pattern where no clause could be matched (incomplete patterns are admissable in Isabelle's logic, see Hupel [9] for a detailed description). This `panic` can also be encountered if an external caller exploited the lossy conversion of sum types as described above and supplied, e.g., a `nil` value as a scrutinee.

Taken together, an entire case expression is translated as a linear sequence of individual clauses, followed by a `panic`:

$$\textsc{stmt}(\text{case } t :: \tau \text{ of } [\overline{p \to b}]) = \overline{\textsc{stmt}(p \to b)}; \texttt{panic("Match failed");}$$

Let us now consider the concrete translation for variable and constructor patterns.

Variable Pattern. We assign the scrutinee t to the variable x to make it available in the scope of b: $\textsc{stmt}(x \to b) = \texttt{\{}x \texttt{ := } \textsc{expr}(t)\texttt{; } \textsc{stmt}(b)\texttt{\}}$.

Constructor Pattern. The pattern is of the form $f[\overline{\tau}_i][\overline{s}_k]$. If all sub-patterns $\overline{s}_k$ are variable patterns, the translation is once again straightforward:

$$\begin{aligned}\textsc{stmt}(f[\overline{\tau}_i][\overline{s}_k] \to b) = \texttt{\{}&q, m \texttt{:=} t\texttt{.(}f[\overline{\tau}_i]\texttt{);}\\ &\texttt{if (}m\texttt{) \{}A_1, \ldots, A_k \texttt{ := } f\texttt{\_dest}(t)\texttt{; } \textsc{stmt}(b)\texttt{\}\}}\end{aligned}$$

Nested constructor patterns are translated in the same way, but pushed inwards into the body of the `if`-statement generated above:

$$\begin{aligned}\textsc{stmt}(f[\overline{\tau}_i][\overline{s}_k] \to b) = \texttt{\{}&q, m \texttt{:=} t\texttt{.(}f[\overline{\tau}_i]\texttt{);}\\ &\texttt{if (}m\texttt{) \{}A_1, \ldots, A_k \texttt{ := } f\texttt{\_dest}(t)\texttt{; } \mathcal{I}\texttt{\}\}}\\ \mathcal{I} = \textsc{stmt}(&\text{case } A_1 \text{ of } s_1 \to (\ldots \to (\text{case } A_k \text{ of } s_k \to b)))\end{aligned}$$

In other words, the sub-patterns are treated as if they were further nested case expressions. This results in a total nesting depth of one level per constructor.

Within the innermost `if`, the body b of the pattern's clause is translated as statement to ensure it returns from the current function.

Optimizing the Nesting Level. The translation described in this section can translate arbitrary patterns, but comes at the price of potentially exponential code blow-up. Even a single pattern consisting of just a constructor and k fields, none of which are proper patterns, will still produce k levels of nested `if`-statements. But if the fields themselves are again data type constructors with sub-patterns, the number of nested levels quickly increases further.

In real-world applications, we can reduce the blow-up by optimizing constructor patterns without arguments. Instead of calling a destructor function, we can emit an equality check, since there are no fields to extract. Multiple equality checks can be joined together using Go's conjunction operator `&&`.

Example. Consider the function hd2 (Fig. 1), which takes a list and returns (optionally) the second element of the list. It is translated into Go as follows:

```
func Hd2[a any] (x0 List[a]) Option[a] {
  if (x0 == (List[a](Nil[a]{}))) {
    return (Option[a](None[a]{}));
  }
  q, m := x0.(Cons[a]);
  if (m) {
    _, c := Cons_dest(q);
    if (c == (List[a](Nil[a]{}))) {
      return (Option[a](None[a]{}));
    }
  }
  q, m := x0.(Cons[a]);
  if (m) {
    _, p := Cons_dest(q);
    q, m := p.(Cons[a]);
    if (m) {
      ya, _ := Cons_dest(q);
      return (Option[a](Some[a]{ya}));
    }
  }
  panic("match failed");
}
```

This piece of generated code benefits from the optimisation described above (in its first two clauses). Also, observe that since unused variables are a compile error in Go, unused bound names above have been generated as _ instead.

4.4 Top-Level Functions

Unlike lambdas that occur within terms, top-level functions in Thingol can have multiple clauses and pattern-match on their arguments, neither of which is supported in Go. It is thus necessary to translate them differently: all equations of the same function will have to be merged, with the pattern matching on their parameters again pushed inwards into the then combined, single function body.

Further, treating them differently from in-term lambda expression also allows the generator to uncurry them, creating code that is much closer to an idiomatic style in Go.

Merging Multiple Clauses. Thingol allows Haskell-style function definition comprising multiple clauses. But in Go, all parameters of functions must be simple variables. Thus, if any of the parameters patterns $\overline{p_i}$ is a proper pattern, a fresh name x_i for it is invented. Likewise, if a parameter is a variable binding instead of a proper pattern, but has multiple different names in two clauses, the name x_i used in the first clause is picked as the name of the parameter in Go.

Pattern Matching. The combined function body then contains a pattern match translation, as described above.[2] Each equation is treated as a clause of a synthetic case-expression; for functions matching on multiple parameters, we again push inwards and translate as if a nested series of case-expressions were present.

Example. Consider this definition for `hd2'`, which is semantically equivalent to `hd2`, but written using multiple equations:

```
fun hd2' :: ∀α.α list ⇒ α option where
  hd2' Nil = None
  hd2' (Cons x Nil) = None
  hd2' (Cons x (Cons y xs) ys) = Some y
```

The generated Go code is identical.

Special Case: Top-Level Constants. Thingol accepts top-level definitions that are not functions, for example: `definition a :: nat where a = 10` . Unfortunately, Go admits top-level variable declarations only for monomophic types, and further disallows function calls in their definitions.

Therefore, we must treat such Thingol definitions as if they were nullary functions. While this changes nothing of the semantics of the translated program, it does incur a (potentially significant) runtime cost: constants will be evaluated each time they are used, instead of only once when the program is initialized.

4.5 Dictionary Construction

On the surface, Isabelle's Haskell-style type classes and Go's interfaces share many of the same features, and are sometimes considered to be near-analogous [3]. However, translating type classes into interfaces does not work, due to an implementation concern: Go directly compiles methods into virtual tables for dynamic dispatch. A Go interface declares multiple *methods*, where each method type must take the generic value as its zeroth (i.e. implicit) parameter. Thingol has no such restriction. Consider, for example:

```
class foo where                    class bar where
  foo :: unit ⇒ α                    bar :: (α ⇒ α) ⇒ unit
```

As Go interfaces, both are invalid: `foo` declares a function whose parameter types do not mention α at all, while `bar`'s function does not take a simple α parameter (but a parameter whose type contains α).

To avoid the additional complexity of treating all these cases separately, we resort to using a dictionary construction [7,9] in all cases. Since the existing SML target of the Code Generator has to deal with the same issue, the required infrastructure is already in place: Thingol's terms come with enough annotations to resolve all type class constraints during translation and replace the implicit instance arguments of functions making use of type classes by explicit dictionary values, which we represent as one data type per type class.

Thus only relatively few things are left to do in Go:

[2] The already-existing Scala target uses a similar transformation.

1. declare a data type for each type class, called its *dictionary* type
2. translate type class constraints on functions into explicit function arguments of dictionary types
3. translate type class instances into either a value of the type class's dictionary type, or, if the instance itself takes type class constraints, to a function producing such a value when given values of dictionary types representing these constraints
4. any time a top-level function is used, the already-resolved type class constraints must be given as explicit arguments

Example. The class declarations (Fig. 1) are translated as follows:

```
type Semigroup[a any] struct {
  Plus func(a, a) a
}

type Monoid[a any] struct {
  Semigroup_monoid Semigroup[a]
  Zero func () a
}

func Sum[a any] (a_ Monoid[a], xs List[a]) a {
  return Fold[a, a](
    func (aa a) func(a) a {
      return func (b a) a { return a_.Semigroup_monoid.Plus(aa, b); };
    },
    xs, a_.Zero()
  );
}
```

4.6 Mapping High-Level Constructs

So far, the shallow embedding we have presented produced code with no dependencies on the Go side, with only the built-in constructs `panic` and `&&` used. All higher-level constructs used by programs (such as lists, numbers) must thus be "brought along" from Isabelle, and are translated wholesale exactly as they are defined in their formalisations. While this guarantees correctness, it is highly impractical for real-world applications: for example, natural numbers as defined in Isabelle/HOL (unary Peano representation, §4.2) require linear memory and quadratic runtime even for simple operations like addition.

Luckily, the Code Generator already has a solution for this conundrum in the form of *printing rules*, which can map Isabelle's types and constants to user-supplied names in the target language. We have set up printing rules mapping:

- Isabelle/HOL's booleans to booleans in Go
- numbers to arbitrary-precision integers (via Go's `math/big` package)
- strings of the `String.literal` type to strings in Go

Unfortunately, linked lists cannot be as easily mapped by default, because Go does not feature a standard implementation of linked lists.

5 Evaluation

Even though Go greatly differs from the existing targets, we have achieved almost full feature parity with the translation described in this paper. Isabelle constructs that are not (cleanly) mapped are:

- infinite data types, which can be defined e.g. via `codatatype` in Isabelle, but are rejected by Go's compiler;
- some low-level string operations that operate on byte values of characters.

Trusted Code Base. All target language generators are part of Isabelle's *trusted code base*, i.e. bugs inside its own code may lead to bugs in the generated program, and cannot be checked for by Isabelle's kernel. Fortunately, our implementation is "just another module" to the core infrastructure; up until Thingol everything remains unchanged, in line with the other language targets.

However, future (more ambitious) code printing may require changes in Thingol: If code printing shall assume more constructs of Go, it would be useful for Thingol itself to have some concept of the syntactic distinction between expressions and statements.

Code Style. The generated Go code is not idiomatic, but neither is the generated code for the other languages. Even though the semantics of SML, OCaml and others may more closely resemble the intention of Isabelle users, the generated code in those languages is also littered with syntactic artifacts. This is evidenced by the fact that neither SML nor OCaml support type classes, and Scala code hardly uses type classes in the way that Haskell does (typically prefering object-orientation). Therefore, we do not envision a future need to align the style of the generated code more closely with the preferred style of hand-written Go code.

The main challenge arises from interfacing between generated and hand-written Go code, both of which would be present in a typical application. For instance, constructing values for the translated `datatype` definitions or using curried functions in Go is unfortunately verbose, and can easily introduce errors.

We therefore recommend to write wrapper code that exposes a "cleaner" interface, ready to be consumed by the real-world application. The wrapper must be written carefully: many explicit type annotations are needed in the code, and not all incorrect type annotations will cause compilation to fail. In particular, if a data type's constructor is annotated with a wrong `interface` type, the assumption underlying the translation of case-expressions will fail, resulting in a "match failed" error at runtime (§4.3).

Another awkward source of problems when integrating the generated code with a larger code base is that Go's standard library lacks common functional data structures, such as lists or tuples (§4.6). Hand-written code would need to deal with the necessary conversions (e.g. from a Go array into a linked list). To some extent, this can be alleviated by leveraging third-party libraries for functional data structures, which are unfortunately not popular in the Go community.

5.1 Case Studies

We conducted two case studies that have confirmed our approach.

Existing Formalisation. At Giesecke+Devrient, we use Isabelle for a substantial formalisation of various graph algorithms powering a financial transaction system. The purpose of the formalisation is to provide real-world security guarantees, such as inability to clone money. We have previously used the Code Generator to produce Scala code as a reference implementation, combined with some hand-written wrapper code and basic unit tests.

As a simple evaluation of Go code generated from the same Isabelle theories, we re-wrote the unit tests and the necessary wrapper code in Go. We obtained equivalent results and could not find bugs in the Code Generator or unintended behaviour of the code it produced. Note that no adaptations of the Isabelle formalisation were necessary, which proves that the Go backend works as a drop-in replacement for the other targets.

Starting from this, we can narrow the formalisation gap mentioned in the introduction. It allows us to link the Isabelle/HOL reference implementation with the real-world production implementation in Go.

HOL-Codegenerator_Test. Isabelle's distribution contains a Code Generator test session which is used as a self-check for the various target languages of the Code Generator. For this paper, a single export command is relevant, which is meant to export a considerable chunk of Isabelle/HOL's library as a stress-test for the Code Generator. This has worked as expected, with the entirety of the test suite successfully compiling in Go.

As a consequence, our approach enables the vast majority of Isabelle users to generate Go code without having to rewrite their formalisation. In particular—because we map to a functional fragment of Go—there is no need for users to reach for a deep embedding of an imperative language.

6 Conclusion

We have presented a translation from Thingol by shallow embedding into a fragment of Go, and implemented it as a target language for Isabelle's code generation framework. The new target language has been used with success to port an existing Isabelle formalisation that was only targeting Scala to additionally target Go. The implementation is readily usable with a standard Isabelle2024 installation and requires merely importing an additional theory file. The suite of existing tests of Isabelle's Code Generator is also supported.

Future Work. The two most promising areas of future work are: leveraging Go's imperative nature by tightly integrating it with Imperative/HOL [2]; and generating code that utilizes more of Go's standard libraries through custom code printing rules. Both can be implemented using similar mechanisms. However, substantial changes to Isabelle's code generation infrastructure are required, because Go demands more type annotations than other target languages.

Acknowledgements. The authors would like to thank Florian Haftmann and Cornelius Diekmann for their contributions to the development. This work has been partially supported by the Federal Ministry of Education and Research (BMBF), Verbundprojekt CONTAIN (13N16582).

Availability. The artifact for this paper is available in the *Archive of Formal Proofs (AFP)* [17] and under the DOI 10.5281/zenodo.11608252.

References

1. Brucker, A.D.: New Code Generator Target: F#. https://mailman46.in.tum.de/pipermail/isabelle-dev/2022-August/017633.html
2. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_14
3. Ellis, S., Zhu, S., Yoshida, N., Song, L.: Generic go to go: dictionary-passing, monomorphisation, and hybrid. Proc. ACM Program. Lang. **6**(OOPSLA2), 1207–1235 (2022)
4. Go Team: the Go programming language specification. https://go.dev/ref/specgo.dev/ref/spec
5. Griesemer, R., et al.: Featherweight Go. Proc. ACM Program. Lang. **4**, 1–29 (OOPSLA) (2020). https://doi.org/10.1145/3428217
6. Haftmann, F.: Code generation from specifications in higher-order logic, Ph.D. thesis, Technische Universität München (2009)
7. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12251-4_9
8. Haftmann, F., Wenzel, M.: Constructive type classes in Isabelle. In: Altenkirch, T., McBride, C. (eds.) Types for Proofs and Programs, pp. 160–174. Springer, Berlin Heidelberg, Berlin, Heidelberg (2007)
9. Hupel, L.: Certifying dictionary construction in Isabelle/HOL. Fund. Inform. **170**(1–3), 177–205 (2019)
10. Jones, S.P.: Haskell 98 language and libraries: the revised report. Cambridge University Press (2003)
11. Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. Theoret. Comput. Sci. **192**(1), 3–29 (1998)
12. Nipkow, T.: Higher-order rewrite systems. In: Hsiang, J. (ed.) Rewriting Techniques and Applications, pp. 256–256. Springer, Berlin Heidelberg, Berlin, Heidelberg (1995)
13. Nipkow, T., Klein, G.: Concrete Semantics. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10542-0

14. Protzenko, J., et al.: Verified low-level programming embedded in F*. Proc. ACM Program. Lang. **1**, 1–29 (ICFP) (2017)
15. Rieu, R.: Development and verification of arbitrary-precision integer arithmetic libraries, Ph.D. thesis, Université Paris-Saclay (2020)
16. Schirmer, N.: A sequential imperative programming language syntax, semantics, hoare logics and verification environment. Archive of Formal Proofs (2008). https://isa-afp.org/entries/Simpl.html
17. Stübinger, T., Hupel, L.: Go Code generation for Isabelle. Archive of Formal Proofs (2024). https://isa-afp.org/entries/Go.html

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Rigorous Floating-Point Round-Off Error Analysis in PRECiSA 4.0

Laura Titolo[1](✉), Mariano Moscato[1], Marco A. Feliu[1], Paolo Masci[1], and César A. Muñoz[2]

[1] Analytical Mechanics Associates Inc., Hampton, USA
{laura.titolo,mariano.m.moscato,marco.feliu, paolo.m.masci}@nasa.gov
[2] NASA Langley Research Center, Hampton, USA
cesar.a.munoz@nasa.gov

Abstract. Small round-off errors in safety-critical systems can lead to catastrophic consequences. In this context, determining if the result computed by a floating-point program is accurate enough with respect to its ideal real-number counterpart is essential. This paper presents PRECiSA 4.0, a tool that rigorously estimates the accumulated round-off error of a floating-point program. PRECiSA 4.0 combines static analysis, optimization techniques, and theorem proving to provide a modular approach for computing a provably correct round-off error estimation. PRECiSA 4.0 adds several features to previous versions of the tool that enhance its applicability and performance. These features include support for data collections such as lists, records, and tuples; support for recursion schemas; an updated floating-point formalization that closely characterizes the IEEE-754 standard; an efficient and modular analysis of function calls that improves the performances for large programs; and a new user interface integrated into Visual Studio Code.

1 Introduction

Round-off errors arise from the difference between real numbers and their finite precision representations. In a floating-point program, round-off errors accumulate throughout the computation. This may lead to a large divergence between the result computed using floating-point arithmetic and the one ideally obtained using real-number arithmetic. In application domains such as avionics, even small rounding errors may have catastrophic consequences if they are not carefully accounted for. Examples of these errors have been found, for instance, in geofencing applications [29] and position encoding algorithms [41]. Several tools have been proposed over the years to reason about floating-point errors (see [8] for an overview). However, most of the proposed tools either target straight-line code and scalar values only, or do not provide sufficient formal guarantees. This limits their applicability to safety-critical real-world applications.

© The Author(s) 2025

This paper presents PRECiSA 4.0, an open-source[1] tool for automatic floating-point round-off error analysis. PRECiSA computes a sound and accurate estimation of the round-off error that may occur in a floating-point program. It supports a large variety of mathematical operators and programming language constructs, including conditionals, let-in expressions, and bounded loops. In addition, PRECiSA automatically generates formal certificates that can be externally checked in the Prototype Verification System (PVS) [33]. These certificates provide formal guarantees on the soundness of the computed round-off error bounds.

An overview of PRECiSA is presented in Sect. 2. PRECiSA 4.0 adds the following features with respect to previous versions of the tool [28,39]:

- A novel *modular analysis* for *function calls* has been implemented (Sect. 3). The user can choose to apply an abstraction on the computed round-off error expressions for function calls to speed up the analysis execution time. This abstraction has been shown to be effective in the analysis of large programs with multiple function calls.
- Support for *data collections* such as lists, records, and tuples has been implemented. In addition, to operate on these collections, native support for the *map* and *fold* recursion schemas has been added (Sect. 4). This new feature avoids the task of manually unfolding the program, resulting in a less error-prone and more efficient analysis of data collections.
- Support for a new *floating-point formalization* has been added. This formalization faithfully characterizes the IEEE-754 standard [24], including special values such as NaN, signed zeros, and infinities (Sect. 5).
- VSCode-PRECiSA, a new user interface integrated into Visual Studio Code, has been added to the PRECiSA distribution (Sect. 6). Besides providing an intuitive way of presenting the analysis results, VSCode-PRECiSA automatizes and simplifies different kinds of tasks such as comparative, sensitivity, and interval analysis. In addition, VSCode-PRECiSA features a new graphical visualization of the values that may cause *conditional instability*. This phenomenon occurs when floating-point round-off errors impact the evaluation of Boolean expressions in conditional guards, thus affecting the control-flow of a program.

In addition to presenting these new features, an experimental evaluation comparing PRECiSA with other state-of-the-art tools is presented in Sect. 7 together with a discussion on related work.

2 PRECiSA

The round-off error of the floating-point expression $\widetilde{op}(\tilde{v}_i)_{i=1}^{n}$ with respect to the real-valued expression $op(r_i)_{i=1}^{n}$, where $\widetilde{op}$ is a floating-point operator representing a real-valued operator op and $\tilde{v}_i$ is a floating-point value representing a real

[1] https://github.com/nasa/PRECiSA.

value r_i, for $1 \le i \le n$, is given by a combination of (i) the propagation of the errors carried out by the arguments $\tilde{v}_i$, and (ii) the error introduced by the application of $\widetilde{op}$ versus op. Throughout this paper, floating-point variables, operands, and expressions are denoted with a tilde on top. The IEEE-754 standard [24] states that every basic operation should be performed as if it were calculated with infinite precision and then rounded to the nearest floating-point value. Thus, the following inequality is assumed to hold for an n-ary floating-point operator $\widetilde{op}$.

$$|R\,(\widetilde{op}(\tilde{v}_i)_{i=1}^{n}) - op(R\,(\tilde{v}_i))_{i=1}^{n}| \le \tfrac{1}{2} ulp\,(op(R\,(\tilde{v}_i))_{i=1}^{n})\,, \tag{2.1}$$

where R is the projection from floats to reals, and the function $ulp\,(r)$ (unit in the last place), for a given real number r, measures the distance between the two consecutive floating-point numbers f_1 and f_2 such that $f_1 < r \le f_2$. The round-off error of a real-valued expression can be bounded as

$$|R\,(\widetilde{op}(\tilde{v}_i)_{i=1}^{n}) - op(r_i)_{i=1}^{n}| \le \varepsilon_{op}(r_i, e_i)_{i=1}^{n} + \tfrac{1}{2} ulp\,(op(R\,(\tilde{v}_i))_{i=1}^{n})\,, \tag{2.2}$$

where $\varepsilon_{op}(r_i, e_i)_{i=1}^{n}$ represents an overestimation of the difference between the application of the real operator on real values and the application of the same operator on the floating-point arguments, and each e_i is a positive real-valued expression modeling an upper bound of the round-off error carried by the floating-point arguments $\tilde{v}_i$ representing the real-valued expression r_i, i.e., $|R\,(\tilde{v}_i) - r_i| \le e_i$.

PRECiSA assumes compliance with the IEEE-754 standard and uses the round-off error model of Formula (2.2) for correctly rounded operators. Dedicated error approximations are defined for a wide variety of mathematical operators, including arithmetic operators, square root, trigonometric functions, exponential and logarithmic functions, floor, and ceiling. For each of these operators, an error expression $\epsilon_{op}(r_i, e_i)_{i=1}^{n}$ is defined as a function of the real-valued operands and corresponding errors such that

$$\epsilon_{op}(r_i, e_i)_{i=1}^{n} \ge \varepsilon_{op}(r_i, e_i)_{i=1}^{n} + \tfrac{1}{2} ulp\,(op(\mathrm{R}fpvi)_{i=1}^{n})\,. \tag{2.3}$$

PRECiSA accepts as input a floating-point program P, which consists of a set of function declarations in the language of PVS, and initial ranges for the input variables, and it computes a correct overestimation of the round-off error that may occur for each function in P. An overview of the PRECiSA workflow is depicted in Fig. 1. PRECiSA first performs a static analysis on the input program by computing the abstract semantics defined in [39]. For every function declaration $\tilde{f}(\tilde{x}_i)_{i=1}^{n}$ in the input program, PRECiSA computes a set of *conditional error bounds* of the form $\langle \eta, \widetilde{\eta} \rangle \twoheadrightarrow (r, e)$, where η is a Boolean expression on reals, $\widetilde{\eta}$ is a Boolean expression on floats, and r, e are real-valued symbolic arithmetic expressions. Intuitively, $\langle \eta, \widetilde{\eta} \rangle \twoheadrightarrow (r, e)$ indicates that if the conditions η and $\widetilde{\eta}$ are satisfied, the result of evaluating $\tilde{f}(\tilde{x}_i)_{i=1}^{n}$ using exact real-number arithmetic is r and the round-off error of the floating-point implementation is bounded by e. The error expression e is built compositionally following the structure of the function body. The Boolean expressions η and $\tilde{\eta}$ model the information on

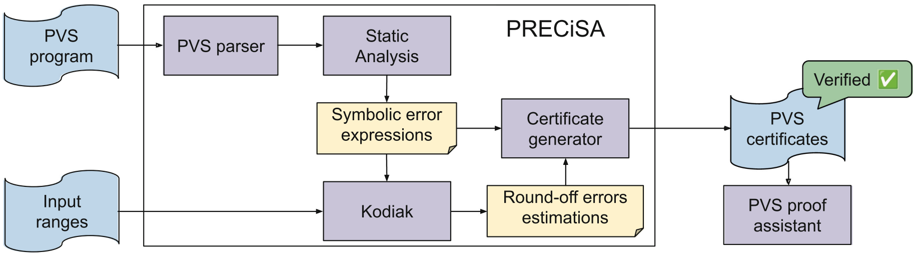


Fig. 1. PRECiSA workflow.

the control flow of the program (i.e., the path conditions from the if-then-else constructs) and the additional restrictions needed when the operators are not total. For example, when dealing with the division operation, it is necessary to guarantee that the divisor is not zero.

The static analysis collects information about real and floating-point execution paths separately. Thus, it is possible to quantify the error due to the so-called *unstable conditions*. This phenomenon occurs when the Boolean guard of a conditional statement is affected by round-off errors. In this case, the real and floating-point Boolean evaluation may be different, causing the control-flow of the floating-point implementation to diverge with respect to its ideal real-number counterpart. An abstraction technique has been introduced in [39] to mitigate the state explosion resulting from the sound treatment of conditional statements. This abstraction collapses the information on the conditional error bounds by keeping separated stable and unstable cases. This way, the accuracy of the error analysis is preserved while the size of the state space is reduced.

Example 1. Consider the function $\widetilde{tcoa}$ that computes the time to co-altitude of two aircraft whose relative altitude is $\tilde{s}$ and relative vertical speed is $\tilde{v}$.

$$\widetilde{tcoa}(\tilde{s},\tilde{v}) = \texttt{ if } \tilde{s} \mathbin{\tilde{*}} \tilde{v} < 0 \texttt{ then } -(\tilde{s} \mathbin{\tilde{/}} \tilde{v}) \texttt{ else } -1, \tag{2.4}$$

PRECiSA computes a set of four different conditional error bounds:

$$\{\langle s*v<0 \wedge v \not= 0, \tilde{s} \mathbin{\tilde{*}} \tilde{v} < 0 \wedge \tilde{v} \not= 0\rangle \twoheadrightarrow (-s/v, \epsilon_/(s,v,e_s,e_v)), \tag{2.5}$$
$$\langle s*v \geq 0, \tilde{s} \mathbin{\tilde{*}} \tilde{v} \geq 0\rangle \twoheadrightarrow (-1,0), \tag{2.6}$$
$$\langle s*v<0 \wedge v \not= 0, \tilde{s} \mathbin{\tilde{*}} \tilde{v} \geq 0\rangle \twoheadrightarrow (-s/v, |-s/v-1|), \tag{2.7}$$
$$\langle s*v \geq 0 \wedge v \not= 0, \tilde{s} \mathbin{\tilde{*}} \tilde{v} < 0 \wedge \tilde{v} \not= 0\rangle \twoheadrightarrow (-1, |s/v-1+\epsilon_/(s,v,e_s,e_v)|)\}. \tag{2.8}$$

The real-valued variables s and v represent the real values of $\tilde{s}$ and $\tilde{v}$, respectively, while e_s and e_v are two positive real variables representing the round-off error of $\tilde{s}$ and $\tilde{v}$, respectively. Formula (2.5) and Formula (2.6) correspond to the cases where real and floating-point computational flows coincide. In Formula (2.5), the negation operator does not contribute to the rounding error, and the following symbolic round-off error expression is computed for the division:

$$\epsilon_/(s,v,e_s,e_v) = \frac{|v|e_s + |s|e_v}{v^2 - |v|e_v} + \tfrac{1}{2} ulp\left(\frac{|s|+e_s}{|v|-e_v}\right). \tag{2.9}$$

In Formula (2.6), the error is 0 since the output is an integer constant. Formula (2.7) and Formula (2.8) model the unstable paths. In these cases, the error is the difference between the output of the two branches taking into account the round-off error of the floating-point result.

The described static analysis is purely compositional, i.e., no assumption is made on the values of the input variables, and the error expressions are composed in a modular fashion. Given initial ranges for the input variables, PRECiSA uses Kodiak[2], a rigorous global optimizer, to compute a sound enclosure of the maximum of the symbolic error expression e. Kodiak implements a formally verified branch-and-bound algorithm presented in [32]. This branch-and-bound algorithm relies on enclosure functions for mathematical operators. These enclosure functions compute provably correct over- and under- approximations of the symbolic error expressions using either interval arithmetic or Bernstein polynomial basis. The algorithm recursively splits the domain of the function into smaller subdomains and computes an enclosure of the original expression in these subdomains. The recursion stops when a precise enclosure is found, based on a given precision, or when a given maximum recursion depth is reached. Both precision and maximum recursion depth can be specified as parameters in PRECiSA. Increasing the value of these parameters will likely improve the accuracy of the analysis but may also increase the execution time. The output of Kodiak is a numerical enclosure for each symbolic error expression. When a function $\tilde{f}$ is associated with more than one conditional error bound, e.g., in the case of conditionals, the overall round-off error of $\tilde{f}$ is defined as the maximum of all the error expressions.

To provide formal guarantees on the analysis results, PRECiSA generates proof certificates ensuring that the round-off error estimations are correct. PRECiSA relies on the higher-order logic interactive theorem prover PVS and a floating-point round-off error formalization included in the NASA PVS Library.[3] More details on this formalization will be given in Sect. 5. For each function, the information in the conditional error bounds is encoded as a PVS lemma stating that, provided the conditions are satisfied and the input variables are in the given numerical ranges, the difference between the floating-point implementation and the real-number specification is at most the computed error estimation. Automatic strategies have been implemented to check the symbolic error expression correctness and the enclosure computed by Kodiak by executing the formally verified PVS implementation of the branch-and-bound algorithm of [32].

3 Optimized Modular Function Call Analysis

PRECiSA supports the compositional analysis of non-recursive function calls. The analysis works by computing a set of conditional error bounds for each function declaration and building an *interpretation* I mapping each function to

[2] https://github.com/nasa/Kodiak.
[3] https://github.com/nasa/pvslib.

its semantics. When the analysis encounters a function call $\tilde{f}(\tilde{x}_i)_{i=1}^n$, it performs a look-up in the interpretation I. For each conditional error bound associated to the function $\langle \phi, \tilde{\phi} \rangle_t \twoheadrightarrow (r, e) \in I(\tilde{f}(\tilde{x}_i)_{i=1}^n)$, PRECiSA performs a substitution of the formal parameters with the actual ones, computing all the possible combinations. For each actual parameter and each conditional error bound in its semantics, $\langle \phi_i, \tilde{\phi}_i \rangle_{t_i} \twoheadrightarrow (r_i, e_i)$, the following conditional error bound is computed for the function call

$$\langle \phi' \wedge \bigwedge_{i=1}^{n} \phi_i, \tilde{\phi}' \wedge \bigwedge_{i=1}^{n} \tilde{\phi}_i \rangle \twoheadrightarrow (r', e'),$$

where $r' = r[\tilde{x}_i/r_i]_{i=1}^n$, $e' = e[\epsilon_{\tilde{x}_i}/e_i]_{i=1}^n$,$\phi' = \phi[\tilde{x}_i/r_i, \epsilon_{\tilde{x}_i}/e_i]_{i=1}^n$, and $\tilde{\phi}' = \tilde{\phi}[\tilde{x}_i/r_i, \epsilon_{\tilde{x}_i}/e_i]_{i=1}^n$. More details on the semantics can be found in [39].

This approach guarantees correctness and accuracy for the optimization process since the error expressions of each function call and of its arguments are unfolded in the global error expression. However, such an error expression may become extremely large for programs with multiple and nested function calls.

To overcome this problem, PRECiSA 4.0 implements an alternative abstract semantics for function calls. In this approach, during the symbolic analysis process, when a function call $\tilde{f}(\tilde{x}_i)_{i=1}^n$ is encountered, instead of computing all the combinations and unfolding the semantics of the function, a placeholder is placed in the call site. Then, during the optimization phase of the analysis, this placeholder is replaced with the worst-case round-off error for the function $\tilde{f}$, computed by Kodiak by optimizing the error expression associated to $\tilde{f}$ and obtained from the interpretation I. It is crucial that the global optimization is executed at each calling site with the correct input ranges for the function arguments with respect to the initial range inputs, provided by the user, and the accumulated round-off error of the arguments. To enable this, PRECiSA relies again on the global optimizer Kodiak. For each argument $\widetilde{arg}$, Kodiak computes its range $[l, u]$ by optimizing the real-valued counterpart of the argument expression. To improve efficiency without compromising too much precision, plain interval arithmetic with no branching—setting the maximum depth parameter to 1 in Kodiak—is used in this phase. The symbolic round-off error expression associated to $\widetilde{arg}$ is computed by PRECiSA. The error due to unstable branches is also taken into account in this phase. This error expression is maximized by Kodiak, and the result err is used to enlarge the argument ranges, obtaining $[l - err, u + err]$. This range is the one used to maximize the function's error expression for that specific call site. The symbolic error expression for each function is computed just once when the interpretation I is built and then its numerical value is computed by maximizing it with different input ranges. The proposed abstract semantics may lead to a loss of correlation between the variables and, potentially, to less accurate estimations. Depending on the desired accuracy/efficiency threshold, the user can choose to perform this abstract function call analysis (the default behavior in PRECiSA 4.0) or to use the option that unfolds the semantics of the function calls and arguments.

The optimization of function calls was key for performing a formal analysis of the NASA DAIDALUS library [30]. This library provides a reference implementation of detect-and-avoid capabilities for unmanned aircraft systems intended to keep aircraft safely separated. In [5], the application of a toolchain to extract a formally verified floating-point C implementation of a DAIDALUS module is presented. The extracted code is annotated with program contracts modeling how the round-off error accumulates through the computation and is instrumented to detect conditional instability. PRECiSA is used in this toolchain as a library to compute round-off errors following the approach presented in [42]. To successfully apply the previous version of PRECiSA to the DAIDALUS module, a pre-processing of the input specification was needed. Without this pre-processing, which included a program slicing and several semantics-preserving simplifications, PRECiSA did not terminate after several minutes. This was due to the complexity of the module which features several conditionals, predicates, and function calls. Using the analysis optimization described in this section, the new version of PRECiSA is able to analyze the original DAIDALUS module without the slicing and simplification used in [5]. Figure 2 and Table 1 show the comparison between the original and the abstract analysis for the numerical functions in the DAIDALUS module. In this case study, the function abstraction improves the performance of the analysis without sacrificing the accuracy. In some cases, slightly more accurate estimations are obtained. This may be due to the large size of the unfolded error expressions, for which the branch-and-bound may not be able to reach enough accuracy within the specified maximum depth. For instance, for `vertical_WCV` (see [30]), the unfolding process times out after 5 min.

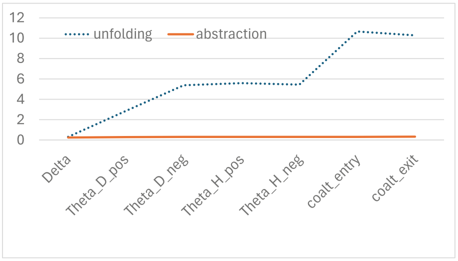


Fig. 2. Times in seconds for the analysis of the DAIDALUS module.

Table 1. Experimental results on the round-off error of the DAIDALUS module.

	unfolding	abstraction
Delta	4.69–14	4.68E-14
Theta_D_pos	1.29E-07	1.07E-07
Theta_D_neg	1.29E-07	1.07E-07
Theta_H_pos	3.55E-15	3.55E-15
Theta_H_neg	3.55E-15	3.55E-15
coalt_entry	8.88E-15	6.66E-15
coalt_exit	3.77E-15	3.77E-15
vertical_WCV	time-out	1.77E-15

4 Data Collections and Bounded Recursion Support

Previous versions of PRECiSA, as well as the majority of floating-point error analysis tools, focus on scalar values. However, it is often the case that safety-critical numerical code makes use of data structures such as lists, tuples, and records. For instance, in the NASA-developed libraries DAIDALUS [30] (aircraft detect-and-avoid) and PolyCARP [31] (polygon computations), a point in space is represented as a tuple (x, y); polygons, used to represent keep-in and keep-out areas such as geofences and weather cells, are represented as lists of points; and aircraft position and velocity vectors are represented as records. These libraries also use bounded loops and typical functional language recursion structures such as map and fold. To enhance the applicability of PRECiSA to these libraries of interest to NASA, support for data collections and the bounded recursion schemas *map* and *fold* have been added to PRECiSA.

Data collections are admitted both as arguments and as return types of functions. Records and tuples are treated in a similar way in PRECiSA. The variable environment used by PRECiSA to store the semantics of local and input variables has been enhanced to accommodate record fields and tuple indices. When a function returns a record or a tuple, PRECiSA performs the static analysis for each element, thus the result is a record or tuple of round-off errors. Furthermore, the structure of the function interpretation I has been modified to support fields and indices. When a function of type record or tuple is called by another function and a field or index is accessed, a lookup in the interpretation is performed as expected.

In contrast to records and tuples, the round-off error of a list is defined as the maximum of the errors of its elements assuming that they are in the same given input range. PRECiSA 4.0 adds support for the following recursion schemas that operate on lists.

$$\texttt{map}\ \tilde{f}\ [l_1, \ldots, l_n] = [\tilde{f}(l_1), \ldots, \tilde{f}(l_n)], \tag{4.1}$$

$$\texttt{fold}\ \tilde{f}\ a\ [l_1, \ldots, l_n] = \tilde{f}(l_1, \ldots (\tilde{f}(l_{n-1}, \tilde{f}(l_n, a))) \ldots). \tag{4.2}$$

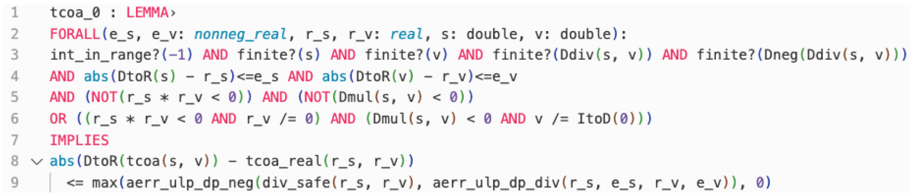
Instead of unrolling the definitions and computing a large error expression, it is sufficient to retrieve the error expression associated to function $\tilde{f}$ in the interpretation, and apply the global optimization process with the correct input variable ranges. For the *map* schema this process is straightforward since all elements in a list are assumed to be in the same input range provided by the user. For the *fold* schema, similar to the function call analysis presented in Sect. 3, it is possible to compute an overestimation of the input ranges in Kodiak. In this phase, n branch-and-bound evaluations of $\tilde{f}$ are performed, where n is the length of the list. The symbolic error expression for $\tilde{f}$ is computed once and then maximized for different values.

5 Floating-Point Formalization

In addition to computing error bounds, an important feature of PRECiSA is the generation of PVS proof certificates that formally ensure that these bounds are correct. PRECiSA relies on the higher-order logic interactive theorem prover PVS [33] and a floating-point formalization originally presented in [6] and extended in [28]. This formalization includes basic definitions related to floating-point numbers, such as their representation, the notion of *ulp*, the notion of subnormal float, and the definition of correctly rounded operators. In addition, it includes a collection of formally verified round-off error estimations for a wide range of mathematical operators. Since PRECiSA's previous release, the PVS floating-point formalization has been restructured and updated to model closely the IEEE-754 standard. To accommodate this change, the certificate generation and the automated proof strategies have been updated in PRECiSA 4.0.

The previous version of PRECiSA assumed that floating-point values were unbounded, meaning that they could be outside the ranges defined by the IEEE-754 standard. Furthermore, special values such as signed zeros, infinities, and NaN were not represented. The new version explicitly introduces bounds for different architectures and special values as defined in the standard. Thus, all floating-point values are required to be either special values or within a valid range.

As an example, Fig. 3 depicts one of the lemmas generated for the function $\widetilde{tcoa}$ from Example 1. Line 2 quantifies over the floating-point variables, `s` and `v`, their real number counterparts, `r_s` and `r_v`, and the non-negative error variables `e_s` and `e_v`. Line 3 states that all the expressions are finitely representable, thus no overflow or NaN can occur. Line 4 states that `e_s` (resp., `e_v`) over-approximates the difference between `r_s` and `s` (resp., `r_v` and `v`). Lines 5–6 specify the Boolean conditions that model the stable conditional error bounds in Formula (2.5) and Formula (2.6). The consequent of the lemma states that the round-off error of $\widetilde{tcoa}$ is at most the maximum between the error of the division $-(\tilde{s}\widetilde{/}\tilde{v})$ and 0, which is the representation error of the value -1. Figure 4 shows a concrete numerical instantiation of the lemma in Fig. 3, which is also automatically generated by PRECiSA. The input ranges are declared in Line 6. The error computed by Kodiak is shown in Line 8 and roughly corresponds to $3.23E-13$. The generated valid range conditions can be used as implicit overflow detectors. In fact, if the value of an expression cannot be proven to be in the range, the lemma cannot be proven. This indicates that an overflow may have occurred.

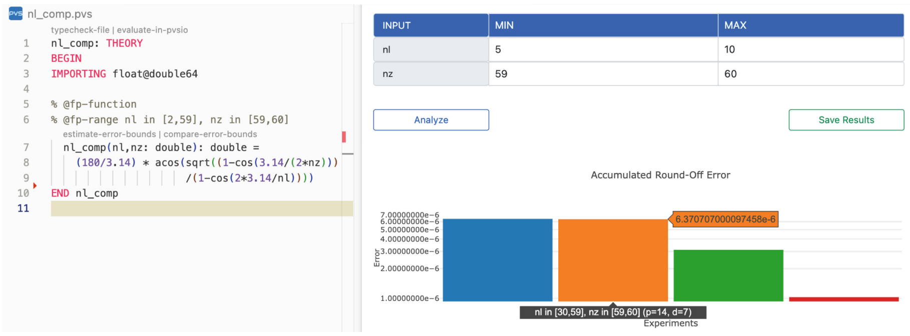


```
1    tcoa_0 : LEMMA›
2    FORALL(e_s, e_v: nonneg_real, r_s, r_v: real, s: double, v: double):
3    int_in_range?(-1) AND finite?(s) AND finite?(v) AND finite?(Ddiv(s, v)) AND finite?(Dneg(Ddiv(s, v)))
4    AND abs(DtoR(s) - r_s)<=e_s AND abs(DtoR(v) - r_v)<=e_v
5    AND (NOT(r_s * r_v < 0)) AND (NOT(Dmul(s, v) < 0))
6    OR ((r_s * r_v < 0 AND r_v /= 0) AND (Dmul(s, v) < 0 AND v /= ItoD(0)))
7    IMPLIES
8  ∨ abs(DtoR(tcoa(s, v)) - tcoa_real(r_s, r_v))
9    | <= max(aerr_ulp_dp_neg(div_safe(r_s, r_v), aerr_ulp_dp_div(r_s, e_s, r_v, e_v)), 0)
```

Fig. 3. Symbolic error lemma in PVS for the $\widetilde{tcoa}$ function.

```
tcoa_c_0 : LEMMA
FORALL(r_s, r_v: real, s: double, v: double):
abs(DtoR(s) - r_s)<=ulp_dp(r_s)/2 AND abs(DtoR(v) - r_v)<=ulp_dp(r_v)/2
AND (NOT(r_s * r_v < 0)) AND (NOT(Dmul(s, v) < 0))
OR ((r_s * r_v < 0 AND r_v /= 0) AND (Dmul(s, v) < 0) AND v /= ItoD(0))
AND r_s ## [|1,300|] AND  r_v ## [|1,300|]
IMPLIES
abs(DtoR(tcoa(s, v)) - tcoa_real(r_s, r_v)) <= 6394759627145231 / 19807040628566084398385987584
```

Fig. 4. Numeric error lemma in PVS for the $\widetilde{tcoa}$ function.



```
nl_comp: THEORY
BEGIN
IMPORTING float@double64

% @fp-function
% @fp-range nl in [2,59], nz in [59,60]
  nl_comp(nl,nz: double): double =
    (180/3.14) * acos(sqrt((1-cos(3.14/(2*nz)))
                    /(1-cos(2*3.14/nl))))
END nl_comp
```

INPUT	MIN	MAX
nl	5	10
nz	59	60

Fig. 5. Round-off error analysis in VSCode-PRECiSA.

6 VSCode-PRECiSA User Interface

VSCode-PRECiSA[4] implements a graphical user interface that integrates PRECiSA into Visual Studio Code, a widely used software development environment developed by Microsoft. Analysis results from PRECiSA are presented using both a bar chart plot diagram (see Fig. 5) and a numerical table. The table presents the numerical results of the analysis along with information on the instability error measuring the divergence of the conditional branches, if applicable, and specific details about the parameters used for the analysis. A series of analysis experiments can be performed for different ranges of input values and combinations of analysis parameters. VSCode-PRECiSA also provides specialized views that facilitate and automate different tasks typically performed with PRECiSA: interval analysis, sensitivity analysis, comparative analysis, and conditional instability analysis.

The *interval analysis* view divides a range of input values into n equally-sized sub-ranges, where n is a positive natural number provided by the user. Floating-point round-off error estimations are computed for each sub-range. The results obtained in this view can be used to gain insights on how to reimplement functions to minimize their round-off errors.

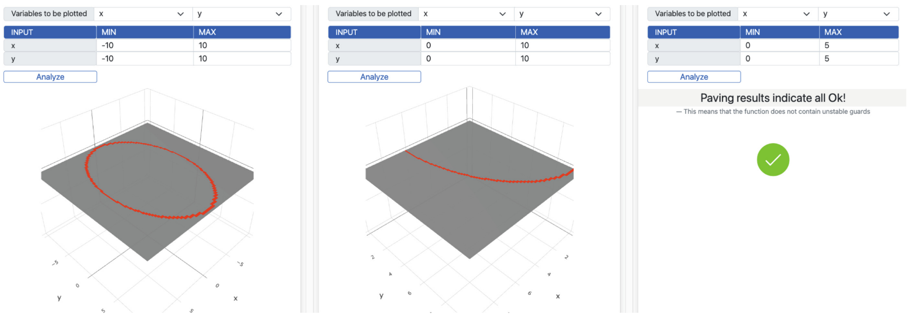
The *sensitivity analysis* view evaluates the floating-point round-off error of a function when the range of input values is affected by a given uncertainty coefficient provided by the user. This view automates the task of checking the

[4] https://github.com/nasa/PRECiSA/tree/master/vscode-precisa.

robustness of a program to round-off errors, i.e., whether small variations of a program's input values lead to unexpectedly large variations in the output.

The *comparative analysis* view shows the floating-point round-off error of two functions evaluated on the same input variables. This view facilitates the assessment of the round-off error in two alternative implementations of an algorithm. The toolkit automatically feeds the two functions with the same input ranges and the analysis results are displayed side-by-side in a bar chart.

As mentioned in Sect. 2, PRECiSA estimates the error associated with unstable conditionals and computes the conditions under which the ideal real number path diverges from the floating-point one. These conditions, called *instability conditions*, are represented by sets of Boolean expressions over both real and floating-point numbers. The *conditional instability analysis* in VSCode-PRECiSA presents visual information on these instability conditions, highlighting which combinations of input variables may alter the control flow of a floating-point program with respect to its ideal real number counterpart. A *2D-mesh* plot is created for a selected pair of variables where the red areas correspond to the regions of possible instability. These regions of instability are computed by the branch-and-bound paving functionality of Kodiak. The paver partitions the input space into regions (called boxes) and uses interval arithmetic to compute the value of the instability conditions over each input region. Due to the over-approximation introduced by interval arithmetic, Kodiak classifies every box as "certainly satisfy," "possibly satisfy," and "certainly do not satisfy." The "possibly satisfy" boxes are progressively refined until a maximum refinement depth or a minimum precision (box size) is reached. The set of boxes that "certainly" and "possibly" satisfy the instability conditions form a sound over-approximation of the inputs that may cause unstable behaviors and, as a consequence, may lead to large computation errors. To the best of the authors' knowledge, PRECiSA is the only tool that supports this kind of analysis. Figure 6 shows the results of the instability analysis for the following function that checks if a point is inside an ellipse-shaped area.



INPUT	MIN	MAX
x	-10	10
y	-10	10

INPUT	MIN	MAX
x	0	10
y	0	10

INPUT	MIN	MAX
x	0	5
y	0	5

Paving results indicate all Ok!

— This means that the function does not contain unstable guards

Fig. 6. Conditional instability analysis in VSCode-PRECiSA.

$$\widetilde{pointInEllipse}(\tilde{x},\tilde{y}) = \texttt{if } \tilde{x} \mathbin{\tilde{*}} \tilde{x} \mathbin{\tilde{/}} 4 \mathbin{\tilde{+}} \tilde{y} \mathbin{\tilde{*}} \tilde{y} \mathbin{\tilde{/}} 9 \leq 10 \texttt{ then } 1 \texttt{ else } -1. \tag{6.1}$$

The figure illustrates that unstable tests may occur for values close to the border of the ellipse, though regions of instability are not always as obvious (see [29] for an example).

7 Related Work

Diverse analysis techniques and tools that estimate the round-off error of floating-point computations have been proposed in the literature.

Gappa [16] computes enclosures for floating-point expressions via interval arithmetic that can be checked in the Coq proof assistant. This method enables a quick computation, but may result in pessimistic error estimations. In Gappa, the bound computation, the certification construction, and their verification may require hints from the user. Thus, some level of expertise is required, unlike PRECiSA, which is fully automatic.

Fluctuat [19] is a commercial analyzer that accepts as input a C program with annotations about input bounds and uncertainties, and it produces bounds for the round-off error of the program expressions. Fluctuat uses a zonotopic abstract domain [21] that extends affine arithmetic [17]. It can soundly identify whether unstable conditional may occur [22] and it provides support for iterative programs by using the widening operators introduced in [18,20]. Unlike PRECiSA, Fluctuat does not produce formal certificates. PRECiSA also implements a widening operator [39], which takes advantage of the information contained in the path conditions of the conditional error bounds to determine when the round-off error of a program may converge. This widening has been applied to simple programs where the error is known to stabilize in a few iterations. More work is needed in this direction to define an effective widening operator for estimating round-off errors for recursive programs.

FPTaylor [37] uses symbolic Taylor expansions to approximate floating-point straight-line expressions and, similar to PRECiSA, applies a global optimization technique to obtain numerical enclosures for round-off errors. It provides support for different rounding modalities such as to-the-nearest, toward infinity, and toward zero. Previous versions of FPTaylor emitted certificates for HOL Light [23], however this functionality appears as deprecated in the last release.

Satire [15] is a tool for estimating round-off errors for straight-line floating-point code with a focus on efficiency. It combines a variant of the technique presented in [37] with an abstraction heuristic that replaces parts of the symbolic error expression with pre-computed constants. Similar to the abstraction presented in Sect. 3, this approach can lead to a loss of correlation between variables and possibly less accurate results, however, it improves the performance of the tool, and it provides a good compromise to scale up to expressions with thousands of operators. In contrast to [37], Satire only computes the first term of the Taylor error expansion. Thus, the computed error bound may not be a sound overestimation. In [1], a sound variation of the abstraction presented in [15], which takes into account also the second-order Taylor term, is presented.

VCFloat [3, 35] is a tool that computes rigorous round-off error terms for straight-line Coq expressions. VCFloat does not generate a Coq certificate, instead the computation of the bound is done entirely within Coq. The input program contains a proof template that needs to be instantiated by the user in order to prove the correctness of the computed bounds.

Daisy [13] is a framework for the analysis and optimization of finite-precision computations. It supports both floating-point and fixed-point arithmetic, and it computes estimations for both absolute and relative errors. Daisy does not generate proof certificates, but the external checker FloVer [4] can be used to validate the bounds computed by Daisy. In [25], Daisy has been enhanced with support for arrays and matrices.

Unlike PRECiSA, which targets programs with common constructs such as let-in constructs, conditional, and function calls, FPTaylor, VCFloat, and Daisy are designed to analyze straight-line program expressions. Table 2 summarizes the features of the above-mentioned tools.

Table 2. Comparison of the features of worse-case round-off error analysis tools.

ïż£

	PRECiSA	FPTaylor	Daisy	VCFloat	Fluctuat	Gappa
proof certificates	✓	✓	✗	✓	✗	✓
conditionals	✓	✗	✗	✗	✓	✗
instability detection	✓	✗	✗	✗	✓	✗
instability analysis	✓	✗	✗	✗	✗	✗
function calls	✓	✗	✗	✗	✓	✗
bounded loops	✓	✗	✗	✗	✓	✗
widening	✓	✗	✗	✗	✓	✗
data collections	✓	✗	✓	✗	✓	✗
rounding modes	✗	✓	✗	✗	✗	✗
fixed-point arith.	✗	✗	✓	✗	✗	✓

Below, PRECiSA 4.0 is compared in terms of accuracy and performance with the following currently maintained open-source tools: Daisy [13] (commit b1705d9), FPTaylor [37] (ver. 0.9.4+dev), and VCFloat2 [3] (commit 10caf1c). This comparison was performed using the standard benchmark suite FPBench [12]. The selected benchmarks involve nonlinear expressions, transcendental functions, and polynomial approximations of functions, taken from equations used in physics, control theory, and biological modeling. These benchmarks and the generated PVS certificates can be found in the PRECiSA distribution. The experimental environment consisted of a 2.6 GHz 6-Core Intel Core i7 with 16 GB of RAM running under MacOS Ventura 13.6.6.

Figure 7 shows numerical round-off error bounds computed by the aforementioned tools. The default configuration is used for each tool. For PRECiSA, Daisy, and FPTaylor, input variables and constants are assumed to be real numbers. This means that they carry an initial round-off error that has to be taken into consideration in the analysis. VCFloat2 does not support the modeling of the initial rounding, thus the input values are assumed to be perfectly representable as a floating-point. This means that the initial rounding error is not taken into account and it is not propagated. Daisy and FPTaylor use the same round-off error model. However, Daisy relies on data-flow analysis and SMT solvers to compute error bounds, while FPTaylor and PRECiSA use global optimization methods. The methods used by FPTaylor and PRECiSA are different, but they coincide on certain operations like sum and multiplication. VCFloat uses interval arithmetic with subdivisions, which may be less accurate than the methods used by FPTaylor and PRECiSA. The times for the computation of the bounds are shown in Fig. 8. The performance of PRECiSA is in line with the other similar tools for most of the examples, and for some of the considered benchmarks PRECiSA is the fastest approach. PRECiSA's times also include the generation of the PVS certificates, while Daisy's include the computation of the relative error bound. In summary, for the considered examples, PRECiSA provides a good trade-off between accuracy and performance together with a wide support for arithmetic operations and programming constructs.

Besides worst-case round-off error analysis tools, other tools have been proposed to improve the quality of floating-point software. The static analyzer Astrée [10] automatically detects the presence of potential floating-point run-time exceptions such as overflows and division-by-zero by means of sound floating-point abstract domains [7,27]. *Precision allocation* (or tuning) tools [2,9,14,36] select the lowest floating-point precision for the program variables that is enough to achieve the desired accuracy. *Program optimization* tools [11,34,38,43] improve the accuracy of floating-point programs by rewriting arithmetic expressions in equivalent ones with a lower round-off error. ReFlow [40], initially distributed as part of PRECiSA, automatically extracts

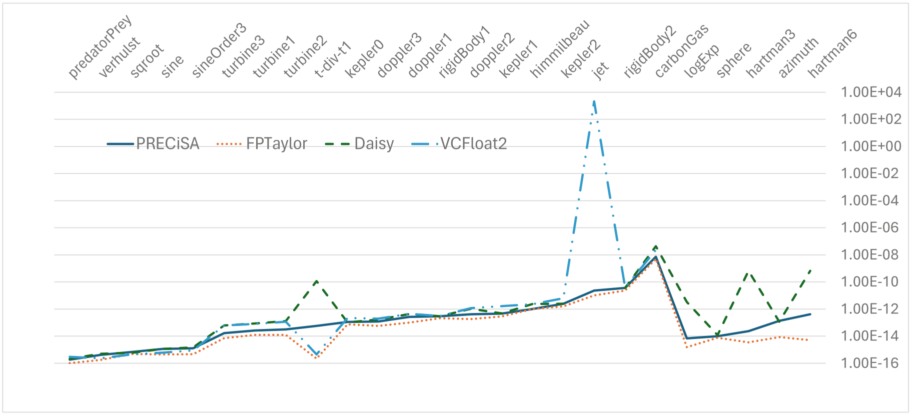


Fig. 7. Experimental results for absolute round-off error bounds.

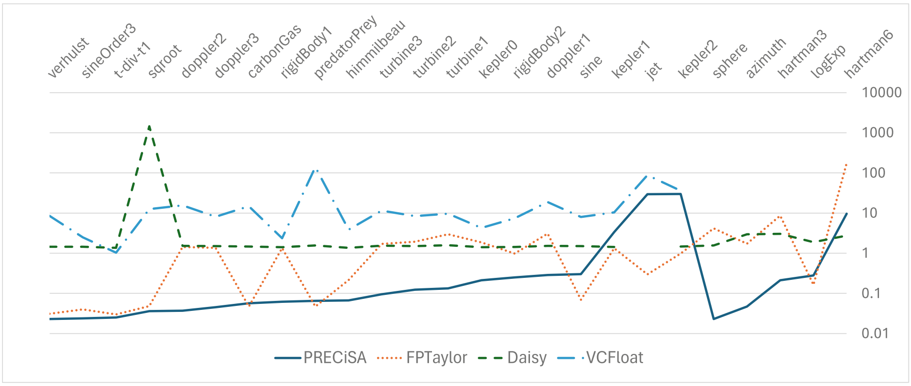


Fig. 8. Times in seconds for the generation of round-off error bounds.

floating-point C code from a PVS real number specification. ReFlow implements a code instrumentation that detects unstable conditionals and annotates the code with contracts that relate the floating-point implementation with the real-valued program specification. The annotated code can be used as input to the static analyzer Frama-C [26]. ReFlow relies on PRECiSA to compute the round-off error estimations and the corresponding PVS proof certificates that guarantee their correctness.

8 Conclusion

This paper presents PRECiSA 4.0, the latest release of a NASA open-source static analyzer for floating-point round-off errors. This version of the tool adds several new features and provides support for a wide range of program constructs and mathematical operators. While the majority of other state-of-the-art round-off error analysis tools are limited to straight-line program expressions, PRECiSA targets programs with function calls, predicates, conditionals, and data structures. Conditional instability analysis is particularly challenging to detect and correct by visual code inspection. Issues related to unstable guards have been discovered in NASA libraries implementing geofencing applications [29] and aircraft detect-and-avoid logics [40]. To the best of the authors' knowledge, the conditional instability analysis presented in this work is the first approach that specifically targets the problem of identifying the source of instability in floating-point programs. PRECiSA 4.0 has been used in several applications at NASA, demonstrating its effectiveness and applicability in real-world problems. PRECiSA is at the core of the floating-point C code generator ReFlow , which has been used to generate formally verified floating-point C code for the NASA libraries DAIDALUS [5] and PolyCARP [29].

In the future, the authors plan to add more features to expand even more the applicability of PRECiSA to real-world programs. For instance, support for fixed-point numbers will be added to enable the analysis of quantized neural networks. The symbolic Taylor error expansion introduced in [37] can be integrated into the analysis performed by PRECiSA. These error approximations can be used as an alternative to, or in combination with, the error expressions implemented in PRECiSA. Additionally, the authors plan to enhance the Kodiak tool to support conditional expressions. This feature will improve the accuracy of the round-off error of conditional if-then-else expressions.

Data Availability Statement. PRECiSA 4.0 is released under NASA Open Source Agreement and it is available at https://github.com/nasa/PRECiSA. Additionally, the companion artifact of this submission can be accessed via the following link: https://doi.org/10.5281/zenodo.12525527.

References

1. Abbasi, R., Darulova, E.: Modular optimization-based roundoff error analysis of floating-point programs. In: 30th International Symposium on Static Analysis, SAS 2023. LNCS, vol. 14284, pp. 41–64. Springer (2023). https://doi.org/10.1007/978-3-031-44245-2_4
2. Adjé, A., Ben Khalifa, D., Martel, M.: Fast and efficient bit-level precision tuning. In: Proceedings of the 28th International Symposium on Static Analysis, SAS 2021. LNCS, vol. 12913, pp. 1–24. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_1
3. Appel, A.W., Kellison, A.: VCFloat2: floating-point error analysis in Coq. In: Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, pp. 14–29. ACM (2024). https://doi.org/10.1145/3636501.3636953
4. Becker, H., Zyuzin, N., Monat, R., Darulova, E., Myreen, M.O., Fox, A.C.J.: A verified certificate checker for finite-precision error bounds in Coq and HOL4. In: 2018 Formal Methods in Computer Aided Design, FMCAD 2018, pp. 1–10. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603019
5. Bernardes Fernandes Ferreira, N., Moscato, M.M., Titolo, L., Ayala-Rincón, M.: A provably correct floating-point implementation of well clear avionics concepts. In: Formal Methods in Computer-Aided Design (FMCAD 2023), pp. 237–246. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_32
6. Boldo, S., Muñoz, C.: A high-level formalization of floating-point numbers in PVS, CR-2006-214298, NASA. Technical report (2006)
7. Chen, L., Miné, A., Cousot, P.: A sound floating-point polyhedra abstract domain. In: Ramalingam, G. (ed.) Programming Languages and Systems, pp. 3–18. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89330-1_2
8. Cherubin, S., Agosta, G.: Tools for reduced precision computation: a survey. ACM Comput. Surv. **53**(2), 33:1–33:35 (2020). https://doi.org/10.1145/3381039
9. Chiang, W., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous floating-point mixed-precision tuning. In: Proceedings of the

44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, pp. 300–315. ACM (2017). https://doi.org/10.1145/3009837.3009846
10. Cousot, P., et al.: The ASTREÉ analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_3
11. Damouche, N., Martel, M.: Salsa: an automatic tool to improve the numerical accuracy of programs. In: 6th Workshop on Automated Formal Methods, AFM 2017, vol. 5, pp. 63–76 (2017). https://doi.org/10.29007/j2fd
12. Damouche, N., Martel, M., Panchekha, P., Qiu, C., Sanchez-Stern, A., Tatlock, Z.: Toward a standard benchmark format and suite for floating-point analysis. In: 9th International Workshop Numerical Software Verification, NSV 2016. LNCS, vol. 10152, pp. 63–77 (2016). https://doi.org/10.1007/978-3-319-54292-8_6
13. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - framework for analysis and optimization of numerical programs (tool paper). In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 270–287. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_15
14. Darulova, E., Kuncak, V.: Sound compilation of reals. In: Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 235–248. ACM (2014). https://doi.org/10.1145/2535838.2535874
15. Das, A., Briggs, I., Gopalakrishnan, G., Krishnamoorthy, S.: An abstraction-guided approach to scalable and rigorous floating-point error analysis. arXiv preprint arXiv:2004.11960 (2020)
16. de Dinechin, F., Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Trans. Comput. **60**(2), 242–253 (2011). https://doi.org/10.1109/TC.2010.128
17. de Figueiredo, L.H., Stolfi, J.: Affine arithmetic: concepts and applications. Numer. Algorithms **37**(1–4), 147–158 (2004). https://doi.org/10.1023/B:NUMA.0000049462.70970.b6
18. Ghorbal, K., Goubault, E., Putot, S.: A logical product approach to zonotope intersection. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 212–226. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_22
19. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 18–34. Springer, Heidelberg (2006). https://doi.org/10.1007/11823230_3
20. Goubault, E., Putot, S.: Perturbed affine arithmetic for invariant computation in numerical program analysis. arXiv preprint arxiv:0807.2961 (2008)
21. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 232–247. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_17
22. Goubault, E., Putot, S.: Robustness analysis of finite precision implementations. In: Shan, C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 50–57. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03542-0_4
23. Harrison, J.: HOL light: an overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 60–66. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_4
24. IEEE: IEEE standard for binary floating-point arithmetic, Technical report, Institute of Electrical and Electronics Engineers (2008)

25. Isychev, A., Darulova, E.: Scaling up roundoff analysis of functional data structure programs. In: Proceedings of the 30th International Symposium on Static Analysis, SAS 2023. LNCS, vol. 14284, pp. 371–402. Springer (2023). https://doi.org/10.1007/978-3-031-44245-2_17
26. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/S00165-014-0326-7
27. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Proceedings of the 13th European Symposium on Programming Languages and Systems, ESOP 2004. LNCS, vol. 2986, pp. 3–17. Springer (2004). https://doi.org/10.1007/978-3-540-24725-8_2
28. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.: Automatic estimation of verified floating-point round-off errors via static analysis. In: Proceedings of the 36th International Conference on Computer Safety, Reliablilty, and Security, SAFECOMP 2017. Springer (2017). https://doi.org/10.1007/978-3-319-66266-4_14
29. Moscato, M., Titolo, L., Feliú, M., Muñoz, C.: Provably correct floating-point implementation of a point-in-polygon algorithm. In: Proceedings of the 23nd International Symposium on Formal Methods, FM 2019. LNCS, vol. 11800, pp. 21–37. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_3
30. Muñoz, C., Narkawicz, A., Hagen, G., Upchurch, J., Dutle, A., Consiglio, M.: DAIDALUS: detect and avoid alerting logic for unmanned systems. In: Proceedings of the 34th Digital Avionics Systems Conference (DASC 2015), Prague, Czech Republic (2015)
31. Narkawicz, A., Hagen, G.: Algorithms for collision detection between a point and a moving polygon, with applications to aircraft weather avoidance. In: Proceedings of the AIAA Aviation Conference (2016)
32. Narkawicz, A., Muñoz, C.: A formally verified generic branching algorithm for global optimization. In: Proceedings of the 5th International Conference on Verified Software: Theories, Tools, Experiments, VSTTE 2013. LNCS, vol. 8164, pp. 326–343. Springer (2013). https://doi.org/10.1007/978-3-642-54108-7_17
33. Owre, S., Rushby, J., Shankar, N.: PVS: a prototype verification system. In: Proceedings of the 11th International Conference on Automated Deduction, CADE 1992. LNCS, vol. 607, pp. 748–752. Springer (1992). https://doi.org/10.1007/3-540-55602-8_217
34. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 1–11. ACM (2015). https://doi.org/10.1145/2737924.2737959
35. Ramananandro, T., Mountcastle, P., Meister, B., Lethin, R.: A unified Coq framework for verifying C programs with floating-point computations. In: Proceedings of CPP 2016, pp. 15–26. ACM (2016). https://doi.org/10.1145/2854065.2854066
36. Rubio-González, C., et al.: Precimonious: tuning assistant for floating-point precision. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, pp. 27:1–27:12. ACM (2013). https://doi.org/10.1145/2503210.2503296
37. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In: Proceedings of the 20th International Symposium on Formal Methods, FM 2015. LNCS, vol. 9109, pp. 532–550. Springer (2015). https://doi.org/10.1007/978-3-319-19249-9_33

38. Thévenoux, L., Langlois, P., Martel, M.: Automatic source-to-source error compensation of floating-point programs. In: 18th IEEE International Conference on Computational Science and Engineering, CSE 2015, pp. 9–16. IEEE Computer Society (2015). https://doi.org/10.1109/CSE.2015.11
39. Titolo, L., Feliú, M.A., Moscato, M., Muñoz, C.A.: An abstract interpretation framework for the round-off error analysis of floating-point programs. In: VMCAI 2018. LNCS, vol. 10747, pp. 516–537. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73721-8_24
40. Titolo, L., Moscato, M., Feliu, M.A., Muñoz, C.A.: Automatic generation of guard-stable floating-point code. In: Dongol, B., Troubitsyna, E. (eds.) IFM 2020. LNCS, vol. 12546, pp. 141–159. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-63461-2_8
41. Titolo, L., Moscato, M., Muñoz, C., Dutle, A., Bobot, F.: A formally verified floating-point implementation of the compact position reporting algorithm. In: Proceedings of the 22nd International Symposium on Formal Methods, FM 2018. LNCS, vol. 10951, pp. 364–381. Springer (2018). https://doi.org/10.1007/978-3-319-95582-7_22
42. Titolo, L., Muñoz, C.A., Feliú, M.A., Moscato, M.M.: Eliminating unstable tests in floating-point programs. In: Mesnard, F., Stuckey, P.J. (eds.) LOPSTR 2018. LNCS, vol. 11408, pp. 169–183. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-13838-7_10
43. Yi, X., Chen, L., Mao, X., Ji, T.: Efficient automated repair of high floating-point errors in numerical libraries. Proc. ACM Program. Lang. **3**(POPL), 56:1–56:29 (2019). https://doi.org/10.1145/3290369

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

FM-Weck: Containerized Execution of Formal-Methods Tools

Dirk Beyer and Henrik Wachowitz

LMU Munich, Munich, Germany

https://gitlab.com/sosy-lab/software/fm-weck

Abstract. Software is ubiquitous in the digital world, and the correct function of software systems is critical for our society, industry, and infrastructure. While testing and static analysis are long-established techniques in software-development processes, it became widely acknowledged only in the past two decades that *formal methods* are required for giving guarantees of functional correctness. Both academia and industry worked hard to develop tools for formal verification of software during the past two decades, with the result that many software verifiers are available now (for example, 59 freely available verifiers for C and Java programs). However, most software verifiers are challenging to find, install, and use for both external researchers and potential users. FM-Weck changes this: It provides a fully automatic, zero-configuration container-based setup and execution for more than 50 software verifiers for C and Java. Both the setup requirements and execution parameters of every supported verifier are provided by the tool developers themselves as part of the FM-Tools metadata format that was established recently and was already used by the international competitions SV-COMP and Test-Comp. With our solution FM-Weck, anyone gets fast and easy access to state-of-the-art formal verifiers, no expertise required, fully reproducible.

Keywords: Formal Methods · Verification · Model Checking · Testing · FM-Tools · Tool Conservation · Reproducibility · Satisfiability Modulo Theories · Provers

1 Introduction

Reliable, correctly functioning IT systems are fundamental in a digital world. One way to achieve correct systems is to apply formal methods. Tools for formal methods are intricate software systems, which often compute abstract models to prove system implementations correct or find errors. There is already a large pool of mature and well-established verification tools (for example, in the area of software verification [1, 2, 3, 4, 5]), and automatic tools are heavily used in industrial software-engineering applications [4, 6, 7, 8]. Sometimes such tools are even used as components in meta verifiers [9, 10, 11, 12, 13, 14]. However, the integration of verification tools provides multiple obstacles: (1) There exists a plentitude of research verification tools that are no longer maintained despite

© The Author(s) 2025

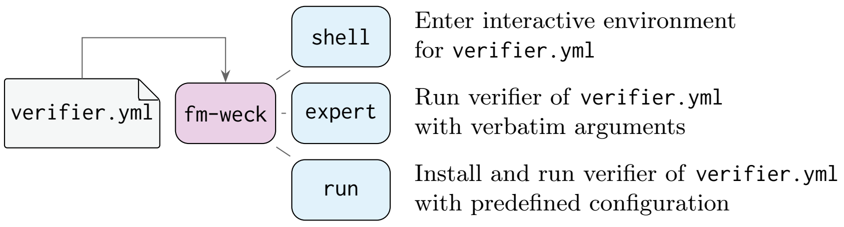

verifier.yml
fm-weck
shell
expert
run
Enter interactive environment for verifier.yml
Run verifier of verifier.yml with verbatim arguments
Install and run verifier of verifier.yml with predefined configuration

Fig. 1: Overview of FM-Weck

delivering interesting results, making them incapable of running in modern software environments, (2) the tools often provide poor documentation of their requirements on the environment (e.g., whether LLVM 9 or LLVM 12 is required), and which operating system they expect (e.g., Ubuntu 20.04), and (3) these tools often have a huge configuration space resulting in a complicated set of command-line interface (CLI) arguments that users have to understand and set correctly. These obstacles deter developers and researchers from experimenting or even integrating verification tools in their own processes and tools [15, 16].

FM-Weck is a command-line tool that mitigates these issues by using the developer-provided metadata from the FM-Tools repository [17, 18]. The FM-Tools repository is a community-maintained source of metadata for formal-methods tools. The repository and the metadata format has been adopted by the international competitions on software verification (SV-COMP) [19] and testing (Test-Comp) [20], and tool developers maintain the information about their tools, including the expected runtime environments and execution parameters. FM-Weck uses these data provided by experts to give researchers and users easy access to controlled runtime environments and execution of more than currently 50 verification tools for C and Java. Figure 1 gives an overview of FM-Weck's three modes of operation, `shell`, `expert`, and `run`, which assist users working with tools for formal methods. In the following section, we briefly introduce the FM-Tools metadata format that is used by the FM-Tools repository (for more details we refer to the format description [17]), then we present how to use FM-Weck's modes of operation before concluding with current applications of the tool.

Related Work. CoVeriTeam [21] is a tool and language for constructing tool compositions. It uses a YAML-based format for the atomic-actor definitions (information where to download, how to assemble command-lines). This format has inspired the format used in FM-Tools. Unfortunately, CoVeriTeam does not configure the execution environment for the tools and simply assumes that the host machine has all required packages readily installed, which FM-Weck solves. Conserving tools for formal methods is an old desire [22], also addressed by CoVeriTeam Service [9]. FM-Weck adds the use of Docker containers to make the environment reproducible and easy to run, also independently from web services.

2 FM-Tools: Tool Metadata

The FM-Tools repository aggregates relevant information about tools for formal methods: It specifies the download location, maintainers, command-line options, as well as other related information. In addition, FM-Tools stores information about container images on which the tool is guaranteed to run according to the maintainers. An FM-Tools file for a specific tool is a YAML document with a precisely defined set of keys (a schema for the metadata of formal-methods tools is available in the repository). FM-Tools was adopted by SV-COMP and Test-Comp in their 2024 edition [19, 20]. As part of FM-Weck, we also provide a Python library [23] that helps users to parse, use, and modify FM-Tools.

```yaml
versions:
  - version: "svcomp24"
    doi: 10.5281/zenodo.10203297
    benchexec_toolinfo_options:
      ["-svcomp24", "-heap", "10000M",
       "-benchmark", "-timelimit", "900 s"]
    required_ubuntu_packages:
      - openjdk-17-jdk-headless
    base_container_images:
      - docker.io/ubuntu:22.04
    full_container_images:
      - registry.gitlab.com/sosy-lab/\
        benchmarking/competition-scripts/user:2024
```

Listing 1: Example of a tool entry in FM-Tools

Listing 1 shows an example of a tool-version entry in FM-Tools. The field `required_ubuntu_packages` specifies the Ubuntu packages that are required to run the tool. The field `base_container_images` specifies the Ubuntu container images on which the required Ubuntu packages can be installed, and with which the tool is guaranteed to run after package installation. The field `full_container_images` specifies self-contained container images that are guaranteed to run the tool out-of-the-box. For a tool t it shall hold that

$$\forall i \in \texttt{base\_container\_images} : \ (i \oplus \texttt{required\_ubuntu\_packages}) \models t \quad (1)$$

$$\forall i \in \texttt{full\_container\_images} : \ i \models t \quad (2)$$

where $\oplus$ denotes the operation of installing the packages on the image i and $i \models t$ denotes that the image i is sufficient to run the tool t.

FM-Tools currently refers to Ubuntu packages, because most tools run on Linux, and Ubuntu as a widespread distribution, whose long-term support keeps specifying and installing the packages straightforward across verifiers.

3 FM-Weck

FM-Weck is a command-line tool written in Python, which consumes FM-Tools [18] tool metadata to execute formal-methods tools inside of a container. (The tool's name is inspired by a German brand of jars for conserving food.) The utility can be used to run, develop, and experiment with formal-methods tools. The software architecture of FM-Weck also allows and encourages usage as a library.

FM-Weck simplifies the execution of formal-methods tools by setting up and starting containers tailored for each tool. FM-Weck can also configure a container runtime such that benchmarks with BenchExec are possible inside of them. To launch the actual container, FM-Weck uses `podman` [24] internally with the `crun` runtime [25]. The FM-Weck CLI comes with three modes of operation: `run`, `expert`, and `shell`.

3.1 FM-Weck Modes

Every command in FM-Weck takes an FM-Tools file as input. This file can be specified either as a path, or as the identifier of the tool. In the latter case, FM-Weck uses the bundled file from the FM-Tools repository with the corresponding name. In any case, users can also specify a specific version of a verifier by appending it with a colon after the file path or name, e.g., `<verifier>:<version>`.

Automatic (`run`) Mode.

```
fm-weck run verifier.yml -p property file.c
```

The `run` mode enables plug-and-play execution of formal-methods tools: it downloads and unpacks a tool from the archive specified in the FM-Tools metadata file ('verifier.yml' above) into a user-specified cache directory on the host system. This cache is mounted into the container, where the verifier is then executed with the given command-line arguments. The `run` mode takes two additional arguments: (1) the `-p` argument specifies a property file, i.e., the goal for the verifier—this can either be a path to the property file or the name of one of the properties used in SV-COMP or Test-Comp, and (2) the files that shall be passed to the tool. In the case of software verifiers, these program files are the input programs to be verified.

Manual (`expert`) Mode.

```
fm-weck expert verifier.yml <args>
```

The `expert` mode is for manual interaction with a verifier: it executes a given verifier, specified through the corresponding FM-Tools YAML file, in its containerized environment passing any additional arguments verbatim to the verifier. Just like in the `run` mode, FM-Weck takes care of downloading and unpacking the verifier as well as setting up the container before the execution. All arguments following the tool `verifier.yml` are passed to the verifier in the container, which makes the `expert` mode essentially act like the verifier if it was executed directly on the host system. The following is an example execution that displays the version of the CPAchecker verifier: `fm-weck expert cpachecker -version`

Interactive (`shell`) Mode.

```
fm-weck shell verifier.yml
```

The `shell` mode enters an interactive shell inside of the container specified by the given verifier. The shell mode launches a Bash shell with the current working directory mounted inside. Users may mount additional directories through a configuration file described in Sect. 3.2. Like with the `expert` mode, the container information is extracted from the FM-Tools metadata file provided by the user. The shell mode takes no additional parameters. The following example starts an interactive shell in the container of Ultimate Automizer: `fm-weck shell uautomizer`

3.2 Project-Specific Configuration

FM-Weck works without any additional configuration, but expert users can still modify aspects of FM-Weck to their needs. Users may set default values and additional files or directories which shall be available inside the container. The configuration is specified in TomL format, as seen in Listing 2. If users define a default image file in this configuration, they can omit the `verifier.yml` in the `shell` mode, and the `*container_image` keys in the `expert` and `run` modes.

```
[defaults]
image = "some_image:latest"
[mount]
"local/path" = "/container/path"
```

Listing 2: Example of a run configuration

Relative paths in the configuration file are relative to the directory that contains the configuration file. If no configuration path is explicitly set via the command line, FM-Weck first looks for a configuration file `.weck` in the current working directory. If this does not exist, it looks for a configuration file `.config/weck` in the user's home directory.

4 Applications

FM-Weck is designed with three core applications in mind: (1) to execute a single tool based on its FM-Tools metadata, (2) to facilitate the execution of unmaintained tools in future competition instances, and (3) as a utility that enables OS-independent execution in CoVeriTeam [21].

4.1 Execution of a Single Tool

FM-Weck provides a bother-free user experience that encourages curious researchers and developers to try and experiment with different verification tools—from well established behemoths to cutting-edge research tools. Users do not have to worry about the tool's dependencies, installation, or complicated command-line configurations. The `run` mode of FM-Weck achieves this goal. Running CPAchecker to find overflows in a C program is as simple as:

```
fm-weck run cpachecker -p no-overflow program.c
```

4.2 Containerized Execution in CoVeriTeam

CoVeriTeam [21] is a framework for cooperative verification. Similar to `fm-weck`, CoVeriTeam takes tool metadata in a YAML format as input, to download and run the tools specified in a cooperative-verification workflow. Each tool is executed inside a containerized environment provided by BenchExec [26]. However, these BenchExec containers do not support OCI container images. This means that all tools running in a CoVeriTeam workflow must be able to run on the host system. We extend CoVeriTeam with an FM-Weck-based run mode. This enables the cooperation of actors regardless of their system requirements.

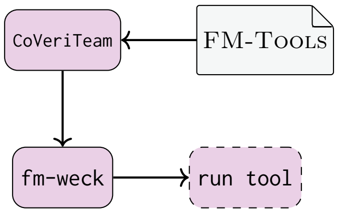


Fig. 2: FM-Weck as executor in CoVeriTeam

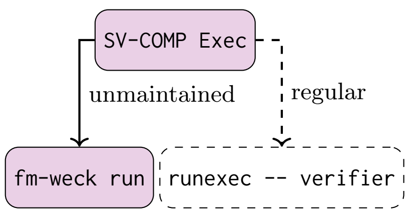


Fig. 3: FM-Weck as drop-in command for SV-COMP infrastructure

Figure 2 illustrates the integration of FM-Weck in CoVeriTeam. Instead of calling BenchExec, CoVeriTeam calls FM-Weck to instantiate a container for the given tool and execute the assembled command inside of it. CoVeriTeam is also written in Python and uses FM-Weck directly as a library.

4.3 Reliable Execution in SV-COMP 2025

SV-COMP comparatively evaluates more than 70 verification tools on an extensive benchmark set [19, 27]. The server infrastructure that executes these millions of verification and validation runs during the competition is hosted on the always-latest Ubuntu LTS Version. This is a formal requirement of the SV-COMP rules. However, there is a growing number of tools that are no longer actively maintained and serve as a retrospective baseline—the so-called hors-concours participants. These tools are benchmarked in the same way as the regular participants, but they do not compete in the ranking. Until SV-COMP 2024, these tools were manually migrated by volunteers to still work on the latest Ubuntu LTS Version, but with 26 hors-concurs participants, the amount of migration-labor becomes infeasible.

With FM-Weck we extend the functionality of the current SV-COMP infrastructure to execute these tools in the SV-COMP 2024 environment. Figure 3 illustrates how FM-Weck is used as a drop-in solution. We wrap the existing invocation of the benchmark command inside of a pre-built image. This image replicates the OS and installed packages of SV-COMP 2024. By default, BenchExec cannot run inside of another container: FM-Weck also sets up the container runtime such that BenchExec works inside of it.

5 Conclusion

We developed FM-Weck, a utility to run formal-methods tools in containerized environments. The goals are to (a) conserve the tools, such that they stay executable in the future, and (b) make it easy for researchers, practitioners, and educators to use and explore the existing tools for formal methods. The application scenarios in CoVeriTeam and SV-COMP infrastructure demonstrate the capabilities of FM-Weck as a library as well as a command-line tool. The tool is open source, licensed under Apache 2.0, and available on GitLab [28].

Data-Availability Statement. The metadata are available in the FM-Tools repository [18] and the source code in the FM-Weck repository [28]. A refined version [29] of the artifact submitted for evaluation [30] is available on Zenodo.

Funding Statement. FM-Weck was funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY).

References

1. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
2. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate automizer and the abstraction of bitwise operations (competition contribution). In: Proc. TACAS (3). pp. 418–423. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_31
3. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Trtík, M., Zaoral, L., Ayaziová, P., Strejček, J.: Symbiotic 10: Lazy memory initialization and compact symbolic execution (competition contribution). In: Proc. TACAS (3). pp. 406–411. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_29
4. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
5. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: The Goblint approach. In: Proc. ASE. pp. 391–402. ACM (2016). https://doi.org/10.1145/2970276.2970337
6. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Proc. IFM. pp. 1–20. LNCS 2999, Springer (2004). https://doi.org/10.1007/978-3-540-24756-2_1
7. Cook, B.: Formal reasoning about the security of Amazon web services. In: Proc. CAV (2). pp. 38–47. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_3
8. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. PLDI. pp. 196–207. ACM (2003). https://doi.org/10.1145/781131.781153
9. Beyer, D., Kanav, S., Wachowitz, H.: CoVeriTeam Service: Verification as a service. In: Proc. ICSE, companion. pp. 21–25. IEEE (2023). https://doi.org/10.1109/ICSE-Companion58688.2023.00017
10. Beyer, D., Lemberger, T., Wachowitz, H.: Reproduction package for TACAS 2024 submission 'Continuous verification: Mitigations of tool restarts for java verifiers'. Zenodo (2023). https://doi.org/10.5281/zenodo.8383787
11. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier (competition contribution). In: Proc. TACAS (3). pp. 365–370. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_22
12. Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
13. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x
14. He, F., Sun, Z., Fan, H.: Deagle: An SMT-based verifier for multi-threaded programs (competition contribution). In: Proc. TACAS (2). pp. 424–428. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_25

15. Alglave, J., Donaldson, A.F., Kröning, D., Tautschnig, M.: Making software verification tools really work. In: Proc. ATVA. pp. 28–42. LNCS 6996, Springer (2011). https://doi.org/10.1007/978-3-642-24372-1_3
16. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: Proc. FMICS. pp. 3–69. LNCS 12327, Springer (2020). https://doi.org/10.1007/978-3-030-58298-2_1
17. Beyer, D.: Conservation and accessibility of tools for formal methods. In: Proc. Festschrift Podelski 65th Birthday. Springer (2024), https://www.sosy-lab.org/research/pub/2024-Podelski65.Conservation_and_Accessibility_of_Tools_for_Formal_Methods.pdf
18. Beyer, D.: Formal-methods tools repository. https://gitlab.com/sosy-lab/benchmarking/fm-tools (2023), accessed: 2024-04-10
19. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15
20. Beyer, D.: Automatic testing of C programs: Test-Comp 2024. In: TBA. Springer (2024)
21. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
22. Steffen, B., Margaria, T., Braun, V.: The Electronic Tool Integration platform: Concepts and design. STTT **1**(1-2), 9–30 (1997). https://doi.org/10.1007/s100090050003
23. Beyer, D., Wachowitz, H.: `lib-fm-tools` repository. https://gitlab.com/sosy-lab/software/lib-fm-tools (2024), accessed: 2024-07-01
24. Podman. https://github.com/containers/podman, accessed: 2023-02-09
25. `crun` runtime. https://github.com/containers/crun (2024), accessed: 2024-04-26
26. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
27. Collection of verification tasks. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks, accessed: 2023-04-01
28. Beyer, D., Wachowitz, H.: FM-Weck repository. https://gitlab.com/sosy-lab/software/fm-weck (2024), accessed: 2024-07-01
29. Beyer, D., Wachowitz, H.: Reproduction package for the FM 2024 article 'FM-Weck: Containerized execution of formal-methods tools'. Zenodo (2024). https://doi.org/10.5281/zenodo.12606323
30. Beyer, D., Wachowitz, H.: Reproduction package for the FM 2024 submission 'FM-Weck: Containerized execution of formal-methods tools'. Zenodo (2024). https://doi.org/10.5281/zenodo.12205513

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

DFAMiner: Mining Minimal Separating DFAs from Labelled Samples

Daniele Dell'Erba[1], Yong Li[1,2](✉), and Sven Schewe[1]

[1] Department of Computer Science, University of Liverpool, Liverpool, UK
{Daniele.Dell-Erba,Yong.Li3,Sven.Schewe}@liverpool.ac.uk

[2] Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract. We propose DFAMiner, a passive learning tool for learning minimal separating deterministic finite automata (DFA) from a set of labelled samples. Separating automata are an interesting class of automata that occurs generally in regular model checking and has raised interest in foundational questions of parity game solving. We first propose a simple and linear-time algorithm that incrementally constructs a three-valued DFA (3DFA) from a set of labelled samples given in the usual lexicographical order. This 3DFA has accepting and rejecting states as well as don't-care states, so that it can exactly recognise the labelled examples. We then apply our tool to mining a minimal separating DFA for the labelled samples by minimising the constructed automata via a reduction to SAT solving. Empirical evaluation shows that our tool outperforms current state-of-the-art tools significantly on standard benchmarks for learning minimal separating DFAs from samples. Progress in the efficient construction of separating DFAs can also lead to finding the lower bound of parity game solving, where we show that DFAMiner can create optimal separating automata for simple languages with up to 7 colours. Future improvements might offer inroads to better data structures.

Keywords: Passive learning · Separating Automata · Three-valued DFA · Parity Game Solving

1 Introduction

The task of inferring a minimum-size separating automaton from two disjoint sets of samples has gained much attention from various fields, including computational biology [21], inference of network invariants [19], regular model checking [26], and reinforcement learning [24]. More recently, this problem has also arisen in the context of parity game solving [6], where separating automata can be used to decide the winner. The breakthrough quasi-polynomial algorithm [8], for example, can be viewed as producing such a separating automaton, and under additional constraints, quasi-polynomial lower bounds can be established,

© The Author(s) 2025

too [8,13]. These applications can be formalised as seeking the minimum-size of DFAs, known as the Min-DFA inference problem, from positive and negative samples.

The Min-DFA inference problem was first explored in [5,18]. Due to its high (NP-complete) complexity, researchers initially focused on either finding local optima through state merging techniques [7,23,29], or investigating theoretical aspects such as reduction to graph colouring problems [11]. Notably, it has been shown that there is no efficient algorithm to find approximate solutions [31].

With the increase in computational power and efficiency of Boolean Satisfiability (SAT) solvers, research has shifted towards practical and *exact* solutions to the Min-DFA inference problem. Several tools have emerged in the literature, including ed-beam/exbar [20], FlexFringe [35], DFA-Inductor [34,36], and DFA-Identify [24].

The current practical and exact solutions to the Min-DFA inference problem typically involve two steps: First, construct the augmented prefix tree acceptor (APTA [12]) that recognises the given samples, and then minimise the APTA to a Min-DFA by a reduction to SAT [20]. Recent enhancements of this approach focus on the second step, including techniques like symmetry breaking [20,34] and compact SAT encoding [20,36]. Additionally, there is an approach on the incremental SAT solving technique specialised for the Min-DFA inference problem, where heuristics for assigning free variables have also been proposed [3]. However, their implementation relies heavily on MiniSAT [17]. We believe that, in order to take advantage of future improvements of SAT solvers, it is better to use a SAT solver as a black-box tool. We note that the second step can be encoded as a Satisfiability Modulo Theories problem [32], which also benefits from our contribution to the first step.

The second step is typically the bottleneck in the workflow. It is known that the number of Boolean variables used in the SAT problem is polynomial in the number of states of the APTA. Smaller APTAs naturally lead to easier SAT problems. This motivates our effort to improve the first step of the inference problem to obtain simpler SAT instances. While previous attempts have aimed at reducing the size of APTAs [7,23,29], we introduce a new and incremental construction of the APTAs that comes with a *minimality* guarantee for the acceptor of the given samples.

Contributions. We propose employing the (polynomial-time) incremental minimal acyclic DFA learning algorithm [14] to extract minimal DFAs from a given set of *positive* samples. More precisely, we extend their algorithm to support the APTA construction from a set of positive samples and a set of negative samples. Notably, the obtained APTA is guaranteed to be the *minimum-size deterministic* acceptor for the labelled sample set S.

We have implemented these techniques in our new tool DFAMiner and compared it with the state-of-the-art tools DFA-Inductor [34,36] and DFA-Identify [24], on the benchmarks generated as described in [34,36]. Our experimental results demonstrate that DFAMiner builds *smaller* APTAs and is therefore significantly faster at finding the Min-DFAs than both DFA-Inductor and DFA-Identify.

To test our technique, we have employed it to extract deterministic safety or reachability automata as witness automata for parity game solving. With DFAMiner, we have established the lower bounds on the size of deterministic safety automata for parity games with up to 7 colours. To the best of our knowledge, this is the *first* time that Min-DFA inference tools have been applied to parity game solving. If they eventually scale, this may lead to new insights into the actual size of the minimal safety automata for solving parity games.

Related work. The learned Min-DFA can be seen as a witness proof that separates the set of good behaviours and the set of bad behaviours for a given system. Therefore, our work can be directly applied to the problems that look for those proofs, such as regular model checking [26] and reinforcement learning [24]. Another standard application is in the active learning of minimal DFAs by equivalence queries [2]. We remark that in [1], non-incremental and incremental constructions were proposed to find small and even minimal APTAs that separate the positive and negative samples. These two constructions are based on state merging techniques of RPNI [29]. Their algorithms are approximate constructions. As a consequence, their constructed APTAs can be smaller (or even larger) than our APTAs, and can no longer be used to extract the minimal separating DFA for S in the second step.

2 Preliminaries

In the whole paper, we fix a finite *alphabet* Σ of letters. A *word* is a finite sequence of letters in Σ. We denote with ε the empty word and with Σ^* the set of all finite words. As usual, we let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. A subset of Σ^* is a *finitary language*. Given a word u, we denote by $u[i]$ the i-th letter of u. We denote by $u[i,k]$ the subword starting at the i-th element and ending at the $(k-1)$-th element when $0 \leq i < k$, and the empty sequence ε when $i \geq k$ or $k = 0$. We denote by $u[i \cdots]$ the word of u starting at the i-th element when $i < |u|$, and the empty sequence ε when $i \geq |u|$. For two given words u and v, we denote by $u \cdot v$ (uv, for short) the concatenation of u and v. We say that u is a *prefix* of w if $w = u \cdot v$ for some word $v \in \Sigma^*$. We denote by $\mathsf{prefixes}(u)$ the set of the prefixes of u. We also extend function $\mathsf{prefixes}$ to a set of words S, i.e. we have $\mathsf{prefixes}(S) = \bigcup_{u \in S} \mathsf{prefixes}(u)$.

Transition system. A *deterministic* transition system (TS) is a tuple $\mathcal{T} = (Q, \iota, \delta)$, where Q is a finite set of states, $\iota \in Q$ is initial state, and $\delta : Q \times \Sigma \to Q$ is a transition function. We also extend δ from letters to words in a usual way, by letting $\delta(q, \varepsilon) = q$ and $\delta(q, a \cdot u) = \delta(\delta(q, a), u)$, where $u \in \Sigma^*$ and $a \in \Sigma$.

Automata. An automaton on finite words is called a deterministic *finite automaton* (DFA). A DFA $\mathcal{A}$ is formally defined as a tuple $(\mathcal{T}, F)$, where $\mathcal{T}$ is a TS, and $F \subseteq Q$ is the set of *accepting* states. DFAs map all words in Σ^* to two values, accepting ($+$) and rejecting ($-$).

A *run* of an DFA $\mathcal{A}$ on a finite word u of length $n \geq 0$ is a sequence of states $\rho = q_0 q_1 \cdots q_n \in Q^+$ such that, for every $0 \leq i < n$, $q_{i+1} = \delta(q_i, u[i+1])$. We write $q_0 \xrightarrow{u} q_n$ if there is a run from q_0 to q_n over u. DFAs have at most one run for each word. A run is *accepting* if it ends in an accepting state $q_n \in F$. A finite word $u \in \Sigma^*$ is *accepted* by $\mathcal{A}$ if it has an accepting run. The set of words accepted by an automaton is called its *language*. The class of words *accepted* by DFAs is known to be regular languages. For a given regular language, the Myhill-Nerode theorem [25,28] helps to obtain the minimal DFA.

DFAs are easy to extend to languages with "don't-care" words.

Definition 1. *A 3-valued DFA (3DFA)*[1] *is defined as a triple* $(\mathcal{T}, A, R)$, *where* $\mathcal{T}$ *is a deterministic TS, and* A, R, *and* $D = Q \backslash (A \cup R)$ *partition the set of states* Q, *where* $A \subseteq Q$ *is the set of* accepting *states;* $R \subseteq Q$ *is the set of* rejecting *states; and the remaining states* D *are called* don't-care *states.*

3DFAs map all words in Σ^* to *three* values: accepting $(+)$, rejecting $(-)$, and don't-care $(?)$, where they are accepting if they have an accepting run, rejecting if they have a rejecting run (which is a run ending in a rejecting state), and don't-care otherwise.

It is possible to identify equivalent words that reach the same state in the minimal 3DFA of a given function $L : \Sigma^* \to \{+, -, ?\}$ [9]. Let x, y be two words in Σ^* and $L \in (\Sigma^* \to \{+, -, ?\})$ be a function. We define an equivalence relation $\sim_L \subseteq \Sigma^* \times \Sigma^*$ as: $x \sim_L y$ if, and only if, $\forall v \in \Sigma^*. L(xv) = L(yv)$.

We denote by $\mid \sim_L \mid$ the index of $\sim_L$, i.e. the number of equivalence classes defined by L. Let $S = (S^+, S^-)$ be a given finite set of labelled samples in Σ^*. We can also see S as a classification function that induces an equivalence relation $\sim_S$. That is, if we set $S^? = \Sigma^* \setminus S$, then $S(u) = \$$ if $u \in S^\$$, where $\$ \in \{+, -, ?\}$. Finally, we conclude with a straightforward proposition that follows from the fact that $\mid \sim_S \mid$ is bounded by $|\mathsf{prefixes}(S)|$.

Fact 1. *Let* S *be a finite set of labelled samples. Then the index of* $\sim_S$ *is finite.*

3 DFAMiner

3.1 Main Problem

Let $S = (S^+, S^-)$ be the given set of labelled samples in the whole paper. Our goal in this paper is to find a *minimal* DFA (Min-DFA) $\mathcal{D}$ for S such that, for all $u \in \Sigma^*$, if $S(u) = \$$, then $\mathcal{D}(u) = \$$, where $\$ \in \{+, -\}$. We call the target DFA a minimal *separating* DFA[2] for S, abbreviated as separating Min-DFA.

Recall that the passive learners for separating Min-DFAs [20,36] usually first construct the APTA $\mathcal{P}$ (and thus a 3DFA) recognising S and then minimise the APTA $\mathcal{P}$ to a Min-DFA using a SAT solver. Our tool DFAMiner follows a similar workflow. The main advantage of DFAMiner compared to prior work is that it has access to an *incremental* construction that produces the minimal 3DFA $\mathcal{M}$

[1] 3DFAs are a standard model for representing positive and negative samples in the literature. In [1], 3DFAs are called deterministic unbiased finite state automata.

[2] The 3DFA that recognises S is called separating DFA for S in [9].

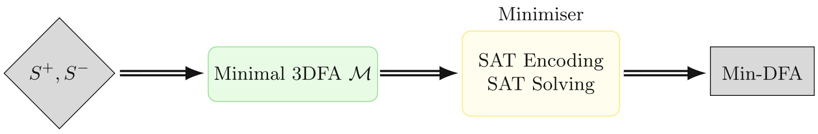


Fig. 1. Workflow of DFAMiner with 3DFAs

of S with respect to $\sim_S$. Furthermore, DFAMiner also supports the use of a DFA pair $(\mathcal{D}^+, \mathcal{D}^-)$ to obtain possibly further reduction on the state space. We call such pair double DFAs.

Definition 2. *A* double DFA *(dDFA) is a tuple* $(\mathcal{T} = \{\mathcal{T}^+, \mathcal{T}^-\}, A, R)$*, where* $\mathcal{T}$ *is the union of two disjoint TSs, and* A*,* R *and* $D = Q \setminus (A \cup R)$ *partition the states* Q *of* $\mathcal{T}$*, such that the languages of* $L^+ = (\mathcal{T}^+, A)$ *and* $L^- = (\mathcal{T}^-, R)$ *are disjoint. We call the words accepted by* L^+ accepting*, the words accepted by* L^- rejecting*, and all other words* don't-care *words.*

Note that since L^+ and L^- are disjoint, every word on $\mathcal{T}$ can have only one accepting run and one rejecting run, although $\mathcal{T}$ has two initial states.

3.2 Workflow Description

Assume that we have an incremental construction of 3DFAs from the given set of samples $S = (S^+, S^-)$. A natural workflow of DFAMiner is to first construct the *minimal* 3DFA $\mathcal{M}$ (which is also a directed acyclic graph) from S and then minimise it using a SAT solver. This approach is depicted in Fig. 1. The components labelled in green or blue in Figs. 1–2 are novel contributions made in our tool. We use the standard SAT-based minimisation approaches of 3DFAs as a black-box [34].

We observe that the minimisation algorithm [34] does not necessarily work only on 3DFAs, but also on dDFAs and even on a pair of nondeterministic finite automata (the encoding will be discussed in Sect. 5). This motivates us to ask the following question: can we construct a dDFA for the pair of samples S? We give a positive answer to this question.

Our construction of dDFAs $\mathcal{N}$ from S is formalised as follows. We construct the minimal 3DFAs $\mathcal{D}^+$ and $\mathcal{D}^-$ that recognise the languages $(S^+, \emptyset)$ and $(S^-, \emptyset)$, respectively, making sure that $\mathcal{D}^+$ and $\mathcal{D}^-$ do not share the same state names. We then combine the two DFAs into a dDFA $\mathcal{N}$, where the initial states of $\mathcal{N}$ are the initial states of both $\mathcal{D}^+$ and $\mathcal{D}^-$, while the transitions between states remain unchanged and we make the accepting states of $\mathcal{D}^+$ and $\mathcal{D}^-$ the accepting and rejecting states of $\mathcal{N}$, respectively. All other states are don't-care states. Note that, although such a dDFA corresponds to two TSs, we can see them as one, since their languages are disjoint. Therefore, even if there are now two initial states, every word will be accepted or rejected by only one of them. The workflow of this construction is depicted in Fig. 2. In this way, we obtain a dDFA $\mathcal{N}$ that recognises exactly the given set S. The empirical evaluation shows

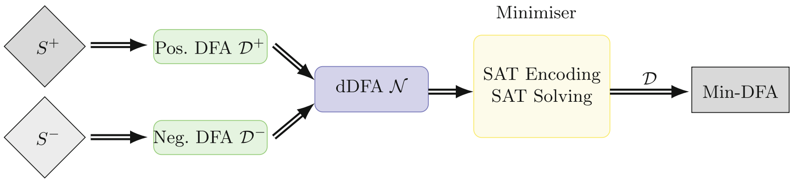


Fig. 2. Workflow of DFAMiner with dDFAs

that the two types of workflows are incomparable (none of them dominates the other in terms of size or speed), hence, both have their place in the learning procedure.

We note that the algorithm for producing dDFAs can be adjusted to produce proper double nondeterministic finite automata (NFAs) when we first translate $\mathcal{D}^+$ and $\mathcal{D}^-$ to NFAs N^+ and N^-, respectively, using standard tools (e.g. [10]) to reduce their size (similar to Fig. 2). The way N^+ and N^- are merged into $\mathcal{N}$ is the same as for $\mathcal{D}^+$ and $\mathcal{D}^-$, and adjusting the SAT encoding to have NFAs (and thus potentially many successors) is straight forward.

For details of the components of DFAMiner, our incremental construction for 3DFAs is reported in Sect. 4, while the SAT-based minimisation algorithm is described in Sect. 5. In Sect. 6 we propose a possible application of DFAMiner in learning minimal separating DFAs by equivalence queries and to parity game solving. We close with an experimental evaluation on standard benchmarks in Sect. 7. A full version of the paper with supplementary materials can be found in [16].

4 Incremental Construction of 3DFAs

4.1 Prior Construction of 3DFAs

Let S be the given labelled sample set and $\mathcal{P}$ the APTA[3] that recognises S constructed with standard procedures [20,24,34,36]. The APTA $\mathcal{P} = (Q, \varepsilon, \delta, F, R)$ is formally defined as a 3DFA where $Q = \mathsf{prefixes}(S)$ is the set of states, ε is the initial state, $F = S^+$ is the set of accepting states, $R = S^-$ is the set of rejecting states, and $\delta(u, a) = ua$ for all $u, ua \in Q$ and $a \in \Sigma$.

Table 1. Size of Min-3DFA and APTA on parity game solving.

$\lvert\Sigma\rvert$	Length	Min-3DFA	APTA
5	7	438	53,277
5	8	541	209,721
5	9	644	835,954
5	10	747	3,369,694
6	7	1279	199,397
6	8	1807	930,870
6	9	2170	4,369,362
6	10	2533	20,689,546

[3] APTAs are called prefix tree unbiased finite state automata in [1] and they are also similar to the prefix-tree Moore Machines in [33].

The main issue is that the size of $\mathcal{P}$ increases dramatically with the growth of the number of samples in S and their length. This is not surprising given that $\mathcal{P}$ maps every word in $\mathsf{prefixes}(S)$ to a unique state.

To show this growth, we have considered samples from parity game solving. Table 1 shows the size comparison between the APTA and its minimal 3DFA (Min-3DFA) representation. With 5 and 6 letters (in this case colours), we can observe that the Min-3DFAs can be much smaller than their corresponding APTA counterparts.

In other words, there are a lot of equivalent states in APTAs that can be merged. To identify equivalent states in $\mathcal{P}$, we can use the equivalence relation $\sim_S$. In fact, since APTAs are acyclic, we can minimise them via a linear-time backward traversal [14]. Further, we show next that we do not have to construct the full APTA $\mathcal{P}$ in order to obtain the Min-3DFA for the given samples.

We will subsequently refer to APTAs constructed by the existing approaches and use 3DFAs for the acceptors constructed by our new technique.

4.2 Incremental Construction of 3DFAs

In [14], an incremental construction of a minimal DFA that accepts a given set of positive samples has been proposed. We extend their algorithm to 3DFAs from a pair $S = (S^+, S^-)$ of sets of labelled samples.

Our algorithm can be seen as the *on-the-fly* version of the combination of the construction of the APTA and its minimisation to the Min-3DFA based on the backward traversal of the APTA. We first describe the minimisation of the APTA tree and then the on-the-fly construction of the Min-3DFAs in the sequel.

For simplicity, let us assume that the *full* APTA tree $\mathcal{P}$ is already given. The crucial step in the minimisation component is to decide whether two states p and q are equivalent. Based on the definition of $\sim_S$, we define that two states $p, q \in Q$ are equivalent, denoted $p \equiv q$ if, and only if:

1. they have the same acceptance status, i.e. they are both accepting, rejecting or don't-care states; and
2. for each letter $a \in \Sigma$, they either both have no successors or their successors are equivalent.

In the implementation, since we only store one representative state for each equivalence class, the second requirement can be simplified as follows:

2'. for each letter $a \in \Sigma$, they either both have no successors or the same successor.

Therefore, it is easy to outline an algorithm to minimise the given APTA tree $\mathcal{P}$ by applying these steps:

1. We first collapse all accepting (respectively, rejecting) states without outgoing transitions to one accepting (respectively, rejecting) state without outgoing transitions, and put the two states in a map *Register*, which allows fast access to their representative states for all states.

2. Then we perform *backward* traversal of states and check if there is a state whose successors are *all* in $Register$. For such states, we identify equivalent states by Rule 2', replace all equivalent states with their representative, and put their representative in $Register$.
3. We repeat Step 2 until all states, including the initial one, are in $Register$.

In this way, we are guaranteed to obtain the Min-3DFA $\mathcal{M}$ that correctly recognises the given set S. Moreover, if we use a hash map for storing all representative states in $Register$, the minimisation algorithm outlined above runs in linear time with respect to the number of states in $\mathcal{P}$. However, as we can see in Table 1, the APTAs can be significantly larger than the corresponding Min-3DFAs. Hence, it is vital to avoid the full construction of the APTA tree $\mathcal{P}$ of S. The key of the *on-the-fly* construction is to identify when a state has been *completely* traversed during construction.

To this end, we need to assume that the samples are already ordered in the usual *lexicographical* order that we will also use to compare the words. That is, the input samples will be first ordered as follows. For two words u and u', we first compare their prefixes of length $\mathsf{min}(|u|, |u'|)$. Then, three cases may arise: one of the two words has a smaller letter than the other at the same position, then that word is smaller; otherwise the two prefixes coincide, and then if the two words have the same same length, u and u' are equal; otherwise one word is longer than the other, then it is greater.

Assume that $S = \{u_1, u_2, \cdots, u_\ell\}$ is ordered. In the process of the creation of the states, we need to detect when a state *cannot* have further successors and then it is ready to be merged with its representative state. Assume that the current 3DFA is $\mathcal{P}_i = (Q_i, \{\iota\}, \delta_i, F_i, R_i)$ and we now input the next sample u_{i+1}. When $i = 0$, $\mathcal{P}_0$ is trivially minimal since $\mathcal{P}_0$ has only a state ι without any outgoing transition. For technical reasons, we let $u_0 = \varepsilon$, which may not appear in the sample set S (note that if there is an empty word ε in S, ι will be set to accepting or rejecting accordingly).

Assume now that $i \geq 0$. We read u_{i+1} and run it on $\mathcal{P}_i$. The sample can be seen as $u_{i+1} = x \cdot y_{i+1}$ with the assumption that $x \in \mathsf{prefixes}(u_{i+1})$ is the *longest* word such that $\delta_i(\iota, x) \neq \emptyset$. Let $p = \delta_i(\iota, x)$. Then, all states along the run of $\mathcal{P}_i$ over x cannot be merged with their representatives, as $\mathcal{P}_i$ requires new states to run the suffix y_{i+1}. Note that x must be a prefix of u_i too, i.e. $x \in \mathsf{prefixes}(u_i)$. This follows from the fact that there must be a run of $\mathcal{P}_i$ over u_i, which is the greatest sample in lexicographic order so far, and every word that has a *complete* run in $\mathcal{P}_i$ must *not* be greater than u_i. In fact, if we assume that x is not a prefix of u_i, then x must be smaller than $u_i[0, |x|]$. This leads to the contradiction that u_i is greater than u_{i+1}. Hence, in this case x must be the empty string. Let $u_i = x \cdot y_i$ and $\rho = p_0 \cdots p_{|u_i|}$ where $p_0 = \iota$ and $p_{|x|} = p$. We can show that all states p_k with $k > |x|$ in the run of $\mathcal{P}_i$ over u_i can be merged with their representative, as they cannot have more (future) reachable states.

If we instead assume that there is a state p_j with $j > |x|$ reached over a future sample u_h with $h > i$, then u_h is smaller than u_{i+1}, which leads to the contradiction that the samples are ordered from the smaller to the bigger. Thus,

Algorithm 1. Incremental construction of the minimal 3DFA from S

```
procedure MAIN PROCEDURE(Sample Set $U$)
    $Register := \emptyset$
    while $U$ has next sample $u$ do
        $x :=$ common_prefix($u$)
        $p := \delta(\iota, x)$                      ▷ the last state over the common prefix $x$
        $y := u[|x| \cdots]$                         ▷ the remaining suffix of $u$
        if has_children($p$) then
            replace_or_register($p$)                 ▷ merge/register all states after $p$
        end if
        add_suffix($p, y$)                           ▷ create run to accept suffix $y$ from $p$
    end while
    replace_or_register($\iota$)                     ▷ merge the run over the last sample
end procedure
procedure REPLACE_OR_REGISTER($p$)
    $r :=$ max_child($p$)                            ▷ obtain the successor over the maximal letter
    if has_children($r$) then                        ▷ $r$ has a successor
        replace_or_register($r$)                     ▷ recursively obtain the run over last sample
    end if
    if $\exists q \in Q.(q \in Register \land q \equiv r)$ then
        max_child($p$) $:= q$                        ▷ merge with its representative
    else
        $Register := Register \cup \{r\}$            ▷ set the 1st state of each class as representative
    end if
end procedure
```

we can identify the representatives for states p_k and merge them in the usual backward manner. It follows that all states except the ones in the run of u_{i+1} in the 3DFA $\mathcal{P}_{i+1}$ are already consistent with respect to $\sim_S$; thus, there is no need to modify them afterwards. After we have input all samples, we only need to merge all states in the run over the last sample u_ℓ with their equivalent states. This way, we are guaranteed to obtain the Min-3DFA $\mathcal{M}$ for S in the end.

The formal procedure of the above incremental construction of the Min-3DFA from S is given in Algorithm 1. Note that, when looking for the run from p over the *last* input sample, we only need to find the successors over the maximal letter by the max_child function. In this way, when we reach the last state r of the run over the last sample (i.e., has_children(r) is false), we can begin to identify equivalent states and replace the successor of p with their representative state q in a backward manner or set the state r as the representative of its equivalent class, as described in the subprocedure replace_or_register. Moreover, in the function add_suffix(p, y), we just create the run from p over y and set the last state to be accepting or rejecting depending on the label of u. In fact, we only extend the the equivalence relation $\equiv$ in replace_or_register [14] to support the accepting, rejecting, and don't-care states, as described before.

Figure 3 depicts all intermediate 3DFAs when running Algorithm 1 on the ordered set $S = \{(000, +), (001, +), (10, -)\}$. Initially, the 3DFA only has the initial state ι without outgoing transitions and *Register* is empty. The algorithm first creates states to accept 000. After receiving sample 001, the algorithm runs the common prefix 00 and merges r with its equivalent states. So, r is added to *Register*. When the sample 10 is read, the common prefix with 001 is ε, then all states after ι in the run over 001 can be merged with their equivalent

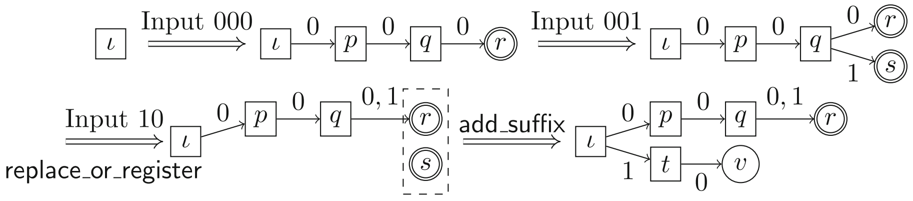


Fig. 3. An example run over $S = \{(000, +), (001, +), (10, -)\}$. Accepting, rejecting and don't-care states are denoted, respectively, by double circles, circles and squares. The dashed rectangle depicts an equivalence class.

states in replace_or_register function. The merge will perform in a backward manner, starting from the last state s until the state p. So, now that we know that also s will not have more successors (because the samples are ordered) we can consider it as *complete* and therefore merge with r. As a consequence, as shown in Fig. 3, all incoming transitions of s are redirected to its representative r and s is deleted. So, $Register = \{r, p, q\}$. After this step, add_suffix will create the new states t and v. Following Algorithm 1, we will eventually obtain the last 3DFA in Fig. 3 as the final result. Note that the biggest intemediate 3DFA constructed by Algorithm 1 is usually much smaller than the full APTA.

Theorem 1. *Let S be a finite labelled set of ordered samples. Algorithm 1 returns the correct Min-3DFA recognising S.*

The proof is basically an induction on the number of input samples and merely extends the intuition described above; we thus omit it here.

DFA construction. The proposed construction of Min-3DFAs is more general than the state-of-the-art incremental one [14] in which it is only checked whether p and q are both accepting or rejecting states when defining the equivalence relation $\equiv$. The other parts of the construction can be modified accordingly.

5 Finding Separating Min-DFAs Using SAT Solvers

This section explains how to extract the separating Min-DFAs from dDFAs[4] built from the 3DFAs through our incremental construction in Sect. 4.2. The encoding approach is used in the Minimiser for both workflows in Figs. 1 and 2 and is agnostic to the SAT solver used. Since minimising DFAs with don't-care words is known to be **NP**-complete [30], it is unlikely to have polynomial-time *exact* algorithm for the second step unless $\mathbf{P} = \mathbf{NP}$.

We assume that we are given a dDFA $\mathcal{N} = (\mathcal{T}, A, R)$, where $\mathcal{T} = (Q, I, \delta)$ is the TS obtained from two DFAs $\mathcal{D}^+$ and $\mathcal{D}^-$. Recall that $\mathcal{T}$ is the TS for the union of $\mathcal{D}^+$ and $\mathcal{D}^-$. In particular, I contains the two initial states from $\mathcal{D}^+$ and $\mathcal{D}^-$. We look for a separating DFA $\mathcal{D}$ of n states for $\mathcal{N}$ such that, for each

[4] 3DFAs can be seen as a special type of dDFAs.

$u \in \Sigma^*$, if $\mathcal{N}(u) = \$$, then $\mathcal{D}(u) = \$$, where $\$ \in \{+, -\}$. Clearly the size of $\mathcal{D}$ is bounded by the size of the TS, i.e. $0 < n \leq |Q|$, since we can obtain a DFA from the dDFA by simply using $\mathcal{D}^+$ (or the complement of $\mathcal{D}^-$). Nevertheless, we aim at finding the *minimal* such integer n.

To do this, we encode our problem as a SAT problem such that there is a separating complete DFA $\mathcal{D}$ with n states if, and only if, the SAT problem is satisfiable. We apply the standard propositional encoding [26,27,34,36]. For simplicity, we let $\{0, \cdots, n-1\}$ be the set of states of $\mathcal{D}$, such that 0 is the initial one. To encode the target DFA $\mathcal{D}$, we use the following variables:

– the transition variable $e_{i,a,j}$ denotes that $i \xrightarrow{a} j$ holds, i.e. $e_{i,a,j}$ is true if, and only if, there is a transition from state i to state j over $a \in \Sigma$, and
– the acceptance variable f_i denotes that $i \in F$, i.e. f_i is true if, and only if, the state i is an accepting one.

Once the problem is satisfiable, from the values of the above variables, it is easy to construct the DFA $\mathcal{D}$. To that end, we need to tell the SAT solver how the DFA should look like by giving the constraints encoded as clauses. For instance, to make sure the resulting DFA is indeed deterministic and complete, we need following constraints:

D1 Determinism:
For every state i and letter $a \in \Sigma$ in $\mathcal{D}$, we have that $\neg e_{i,a,j} \vee \neg e_{i,a,k}$ for all $0 \leq j < k < n$.
D2 Completeness: For every state i and letter $a \in \Sigma$ in $\mathcal{D}$, $\bigvee_{0 \leq j < n} e_{i,a,j}$ holds.

Moreover, to make sure the obtained DFA $\mathcal{D}$ is separating for $\mathcal{N}$, we also need to perform the product of the target DFA $\mathcal{D}$ and $\mathcal{N}$. In order to encode the product, we use extra variables $d_{p,i}$, which indicates that the state p of $\mathcal{N}$ and the state i of $\mathcal{D}$ can both be reached on some word u. The constraints we need to enforce that $\mathcal{D}$ is separating for $\mathcal{N}$ are formalised as below:

D3 Initial condition: $d_{\iota,0}$ is true for all $\iota \in I$. (0 is the initial state of $\mathcal{D}$.)
D4 Acceptance condition: for each state i of $\mathcal{D}$,
D4.1 Accepting states: $d_{p,i} \Rightarrow f_i$ holds for all $p \in A$;
D4.2 Rejecting states: $d_{p,i} \Rightarrow \neg f_i$ holds for all $p \in R$;
D5 Transition relation: for a pair of states i, j in $\mathcal{D}$,
$d_{p,i} \wedge e_{i,a,j} \Rightarrow d_{p',j}$ where $p' = \delta(p, a)$ for all $p \in Q$ and $a \in \Sigma$.

Let $\phi_n^{\mathcal{N}}$ be the conjunction of all these constraints. Then, Theorem 2 follows.

Theorem 2. *Let $\mathcal{N}$ be a dDFA of S and $n \in \mathbb{N}$. Then $\phi_n^{\mathcal{N}}$ is satisfiable if, and only if, there exists a complete DFA $\mathcal{D}_n$ with n states that is separating for $\mathcal{N}$.*

Let n be the minimal integer such that $\phi_n^{\mathcal{N}}$ is satisfiable. Then $\mathcal{D}_n$ is a separating Min-DFA for the sample set S.

The formula $\phi_n^{\mathcal{N}}$ contains $(n^3 \cdot |\Sigma| + n^2 \cdot |Q| \cdot |\Sigma|)$ constraints.

When looking for separating DFAs, the SAT solver may need to inspect multiple isomorphic DFAs that only differ in their state names for satisfiability. If those isomorphic DFAs are not separating for $\mathcal{N}$, then the SAT solver still has to prove this for each DFA. To reduce the search space, DFAMiner uses the technique in [34] to check only a representative DFA for all isomorphic DFAs.

6 Applications

Apart from those mentioned in the introduction, in this section we describe two new applications.

6.1 Active Learning of Separating Min-DFAs

Our tool can be applied to the active learning of minimal DFAs using only equivalence queries (EQs). In fact, minimal DFAs cannot be exactly learned using polynomial number of EQs in the size of the target Min-DFA [2].

While learning, we maintain a growing set of pairs $S_i = (S_i^+, S_i^-)$ to store the positive and negative words. That is, for all the indexes i, $S_{i+1}^+ \supseteq S_i^+$, $S_{i+1}^- \supseteq S_i^-$, and $S_{i+1} \neq S_i$. For each $i > 0$, DFAMiner finds a Min-DFA $\mathcal{D}_i$ for S_i and ask an EQ that the teacher returns a *yes* if $\mathcal{D}_i$ accepts all positive and rejects all negative words in the target language L, or provides one (*positive* or *negative*) counterexample (CEX) u otherwise. We can start with $S_0 = (\emptyset, \emptyset)$ and propose a DFA $\mathcal{D}_0$ accepting nothing, and then obtain S_{i+1} from S_i by adding the CEX u to either S_i^+ or S_i^-. If $\mathcal{D}_i(u) = +$, then u is a negative word since u is misclassified by $\mathcal{D}_i$ and should be added into S_{i+1}^-; otherwise, we add u into S_{i+1}^+. (The other set will remain the same.) Since $\mathcal{D}_i$ is consistent with S_i but not with all S_j with $j > i > 0$, all $\mathcal{D}_i$ are smaller or equal in size to the target DFA $\mathcal{D}$. For some $k > 0$, S_k will uniquely characterise L, $\mathcal{D}_k$ will accept L, where k can be exponential in the number of states in $\mathcal{D}_k$ in the worst case.

6.2 Learning Separating Automata for Parity Games

A few years ago an algorithm to solve parity games in quasi-polynomial time [8] has been proposed. It has then been shown that the underlying approach essentially builds a separating automaton of quasi-polynomial size to distinguish runs with only winning cycles (according to the parity condition) from the losing ones [13]. Such a separating automaton distinguishes the two disjointed languages composed of the set of infinite words that correspond to paths on the graph where the highest colour occurring is *even* (hence, winning), or *odd* (losing). Where each colour occurs only once, a cycle occurs when a colour has repeated at least twice. For instance, the word $(001212)^\omega$ contains only even cycles including 00,121 and 212, while the word $(1312331)^\omega$ contains only odd cycles such as 131, 3123, and 33. Given a parity game $\mathcal{G}$, and a separating automaton $\mathcal{S}$ that accepts only even cycles and rejects odd cycles, solving the parity game $\mathcal{G}$ can be reduced to solving the safety game $\mathcal{G} \otimes \mathcal{S}$ [6]. Although the product game is much bigger than $\mathcal{G}$, safety games are easier to solve than parity games. Moreover, the constructed separating automaton $\mathcal{S}$ is quasi-polynomial in the number of colours, which gives an upper bound for solving parity games.

These separating safety automata work on infinite words, but we will employ our tool to learn them by using finite-length samples. This is because as long as the length of the finite sample words is long enough, the learned DFAs will converge to the correct safety automata. The hardest case for the separation approach [6] occurs when the colours are unique (occur only once, hence, the colour itself can be used as a node identifier, making detection of cycles easier). We have implemented this case as follows: we fix an alphabet with $\mathbf{c}$ different colours, a length $\ell > \mathbf{c}$, (to ensure that each word contains at least one cycle), and $\mathbf{c}$ as highest colour. In the learned DFA, we must accept a word if all cycles are winning (e.g. 001212) and reject it if all cycles are losing (e.g. 13123312). Words with winning and losing cycles (e.g. 21232) are don't-care words.

The resulting automata are always safety automata that reject all words that have not seen a winning cycle after (at most) ℓ steps, as well as some words that have seen both, winning and losing cycles (don't-care word), or, alternatively, reachability automata that accept all words that have not seen a losing cycle after at most ℓ steps (again, except don't-care ones). Thus, the size of the Min-DFA falls when increasing the sample length ℓ, and *eventually stabilises*. Using such a separating automaton reduces solving the parity game to solving a safety game [6].

Separating automata built with the current state-of-the-art construction [8] grow quasi-polynomially, and since it is not known whether these constructions are optimal, we applied DFAMiner to learn the most succinct separating automata for the parity condition.

Table 2 shows the application of DFAMiner to the parity condition up to 7 colours (from 0 to 6). For each maximal colour we report the length required to build the minimal separating automaton, the size of the obtained DFA, and the number of all positive and negative samples generated. Although most words have both wining and losing cycles (don't-care words), the positive and negative samples grow *exponentially*, too, which is why we stopped at 7 colours.

Table 2. Samples required to learn the minimal separating automata for solving parity games.

Colours	2	3	4	5	6
DFA Size	3	3	5	5	9
Length	3	5	7	11	15
#Pos	3	130	1,645	9,375,269	4,399,883,736
#Neg	5	31	5,235	1,009,941	38,871,920,470

While the APTA size constructed by DFA-Inductor grows exponentially, the sizes of dDFAs and 3DFAs seem to grow only constantly when increasing the length of the samples for a fixed colour number. Consequently, *all* versions of DFA-Inductor were only able to solve cases with at most 4 colours, while DFAMiner can manage to solve cases up to 6 colours and length 16. To further push the limit of DFAMiner for parity game solving, we have also provided an efficient SAT encoding for parity games. These supplementary data are provided in [16]. With the constructions for both 3DFAs and dDFAs and the efficient encoding, the bottleneck of the whole procedure is no longer solving the Min-DFA inference problem, but the generation of samples. With a better sample generation approach, we believe that this application can give insights on the structure of minimal safety automata for an arbitrary number of colours.

7 Evaluation

To further demonstrate the improvements of DFAMiner[5] over the state of the art, we conducted comprehensive experiments on standard benchmarks [34, 36]. We compared with DFA-Inductor [36] and DFA-Identify[6] [24], the state of the art tools publicly available for passive learning tasks. Unlike DFAMiner and DFA-Inductor, DFA-Identify uses a SAT encoding of graph coloring problems [20] and the representative DFAs in the second step [34]. Like DFA-Inductor, DFAMiner is also implemented in Python with PySAT [22]. We delegate all SAT queries to the SAT solver CaDiCal 1.5.3 [4] in all tools.

DFAMiner accepts samples formalised in the Abbadingo[7] format.

The experiments were carried on an Intel i7-4790 3.60 GHz processor. In Table 3, each index N reports the results of 100 benchmark instances of random samples. Each benchmark has $50 \times N$ samples. For every index, we show the average time and the percentage of instances solved within 1,200 s. The alphabet for the samples has two symbols while the size of the generated DFA is N. We compare four approaches to inferring Min-DFAs: DFA-Inductor, DFA-Identify, and DFAMiner with both 3DFA (3DFA-MIN) and dDFA (dDFA-MIN). Both dDFA-MIN and 3DFA-MIN perform better than DFA-Inductor and DFA-Identify, on average they are *three* times faster. DFA-Inductor can minimise within 20 min instances up to level 13, while the two variants of DFAMiner can scale one more level and minimise one third of the instances of level 15. On these random samples the dDFA approach is slightly faster than the 3DFA one.

Table 3. Comparison for the minimisation of DFAs from random samples of DFAMiner with DFA inductor.

	DFA-Inductor		DFA-Identify		dDFA-MIN		3DFA-MIN	
N	avg	%	avg	%	avg	%	avg	%
4	0.12	100	0.09	100	0.03	100	0.02	100
5	0.29	100	1.38	100	0.06	100	0.05	100
6	0.67	100	2.33	100	0.30	100	0.18	100
7	1.81	100	4.12	100	0.80	100	0.73	100
8	3.57	100	9.70	100	1.29	100	1.25	100
9	10.84	100	20.76	100	3.83	100	3.78	100
10	50.91	100	44.57	100	17.88	100	16.80	100
11	154.73	100	128.69	100	55.12	100	59.46	100
12	399.52	96	373.65	99	144.27	100	162.39	100
13	850.04	74	785.93	82	390.10	99	418.62	97
14	1125.59	19	1099.92	23	809.88	76	861.10	69
15	1182.98	6	1197.61	1	1060.18	37	1062.02	34
16	1188.17	1	1184.82	3	1167.58	4	1164.02	5

[5] https://github.com/liyong31/DFAMiner.
[6] https://github.com/mvcisback/dfa-identify.
[7] https://abbadingo.cs.nuim.ie/.

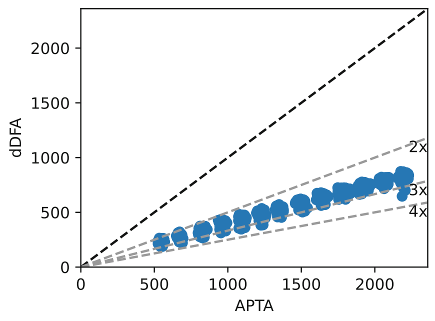


Fig. 4. Scatter plot on automata size

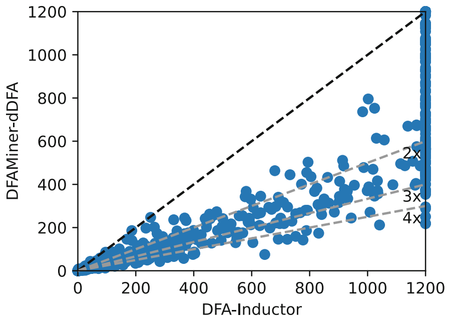


Fig. 5. Scatter plot on runtime (secs)

Figures 4 and 5 report the comparison on the size of the APTA/dDFA (on the left) and minimisation time (on the right) for the previous benchmark. In these two figures, instead of the mean values, we show the individual data for each sample. Both DFA-Inductor and DFA-Identify build the same APTA (they differ for the encoding step), and as shown in Fig. 4, its size is three times larger than the dDFA built by DFAMiner, no matter how big the final DFA is. Figure 5, instead, shows that, when using a dDFA, DFAMiner always performs better than DFA-Inductor, on average three times faster with peaks of more than four times faster. The comparison between dDFA and DFA-Identify is similar.

The experimental results have confirmed that our construction of sample representations significantly advances the state-of-the-art, making it a valuable contribution to the Min-DFA inference problem. We note that DFA-Inductor 2 [36] is faster than DFA-Inductor due to a better encoding of the representative DFAs. Nonetheless, DFAMiner still performs significantly better than DFA-Inductor 2 regarding the overall number of solved cases and running time. For a fair comparison, we choose DFA-Inductor as the baseline, as DFAMiner only differs from it in the construction of APTAs. Additional comparisons on runtime and automata size with DFA-Inductor 2 can be found in [16].

8 Discussion and Future Work

We propose a novel and more efficient way to build APTAs for the Min-DFA inference problem. Our contribution focuses on a compact representation of the the positive and negative samples and, therefore, provides the leeway to benefit from further enhancements in solving the encoded SAT problem.

Natural future extensions of our approach include implementing the tight encoding of symmetry breaking [36]. Another easy extension of our construction is to learn a set of decomposed DFAs [24], thus improving the overall performance as well. A more complex future work is to investigate whether or not one can similarly construct a deterministic Büchi automaton based on ω-regular sets of accepting, rejecting, and don't-care words that provides a minimality guarantee for a given set of labelled samples.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback. This work has been supported in part by the NSFC grant 62102407 and the EPSRC through grants EP/X021513/1, EP/X017796/1, and EP/X03688X/1.

Data Availability. The source code and data are available in [15].

References

1. Alquezar, R., Sanfeliu, A.: Incremental grammatical inference from positive and negative data using unbiased finite state automata. In: Shape, Structure and Pattern Recognition, Proc. Int. Workshop on Structural and Syntactic Pattern Recognition, SSPR, vol.94, pp. 291–300 (1995)
2. Angluin, D.: Negative results for equivalence queries. Mach. Learn. **5**, 121–150 (1990). https://doi.org/10.1007/BF00116034
3. Avellaneda, F., Petrenko, A.: Learning minimal DFA: taking inspiration from RPNI to improve SAT approach. In: Ölveczky, P.C., Salaün, G. (eds.) Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings, LNCS, vol. 11724, pp. 243–256. Springer (2019). https://doi.org/10.1007/978-3-030-30446-1_13
4. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
5. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE Trans. Comput. **21**(6), 592–597 (1972). https://doi.org/10.1109/TC.1972.5009015
6. Bojańczyk, M., Czerwiński, W.: An Automata Toolbox. unpublished (2018)
7. Bugalho, M.M.F., Oliveira, A.L.: Inference of regular languages using state merging algorithms with search. Pattern Recognit. **38**(9), 1457–1467 (2005). https://doi.org/10.1016/J.PATCOG.2004.03.027
8. Calude, C., Jain, S., Khoussainov, B., Li, W., Stephan, F.: deciding parity games in quasipolynomial time. In: Symposium on Theory of Computing 17, pp. 252–263. Association for Computing Machinery (2017)
9. Chen, Y., Farzan, A., Clarke, E.M., Tsay, Y., Wang, B.: Learning minimal separating DFA's for compositional verification. In: Kowalewski, S., Philippou, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings, LNCS, vol. 5505, pp. 31–45. Springer (2009). https://doi.org/10.1007/978-3-642-00768-2_3
10. Clemente, L., Mayr, R.: Efficient reduction of nondeterministic automata with application to language inclusion testing. Log. Methods Comput. Sci. **15**(1) (2019). https://doi.org/10.23638/LMCS-15(1:12)2019
11. Coste, F., Nicolas, J.: Regular inference as a graph coloring problem. In: IWGI (1997)

12. Coste, F., Nicolas, J.: How considering incompatible state mergings may reduce the DFA induction search tree. In: Honavar, V.G., Slutzki, G. (eds.) Grammatical Inference, 4th International Colloquium, ICGI-98, Ames, Iowa, USA, July 12-14, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1433, pp. 199–210. Springer (1998). https://doi.org/10.1007/BFB0054076
13. Czerwinski, W., Daviaud, L., Fijalkow, N., Jurdzinski, M., Lazic, R., Parys, P.: Universal trees grow inside separating automata: quasi-polynomial lower bounds for parity games. In: Symposium on Discrete Algorithms 19, pp. 2333–2349. SIAM (2019)
14. Daciuk, J., Mihov, S., Watson, B.W., Watson, R.E.: Incremental construction of minimal acyclic finite state automata. Comput. Linguistics **26**(1), 3–16 (2000). https://doi.org/10.1162/089120100561601
15. Dell'Erba, D., Li, Y., Schewe, S.: Artifact for DFAMiner: Mining Minimal Separating DFAs from Labelled Samples (Jun 2024). https://doi.org/10.5281/zenodo.12528885
16. Dell'Erba, D., Li, Y., Schewe, S.: DFAminer: mining minimal separating DFAs from labelled samples (2024). https://arxiv.org/abs/2405.18871
17. Eén, N., Sörensson, N.: An extensible sat-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
18. Gold, E.M.: Complexity of automaton identification from given data. Inf. Control. **37**(3), 302–320 (1978). https://doi.org/10.1016/S0019-9958(78)90562-4
19. Grinchtein, O., Leucker, M., Piterman, N.: Inferring network invariants automatically. In: Furbach, U., Shankar, N. (eds.) Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4130, pp. 483–497. Springer (2006). https://doi.org/10.1007/11814771_40
20. Heule, M., Verwer, S.: Exact DFA identification using SAT solvers. In: Sempere, J.M., García, P. (eds.) Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6339, pp. 66–79. Springer (2010). https://doi.org/10.1007/978-3-642-15488-1_7
21. de la Higuera, C.: A bibliographical study of grammatical inference. Pattern Recognit. **38**(9), 1332–1348 (2005). https://doi.org/10.1016/J.PATCOG.2005.01.003
22. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: a python toolkit for prototyping with SAT oracles. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10929, pp. 428–437. Springer (2018). https://doi.org/10.1007/978-3-319-94144-8_26
23. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: Honavar, V.G., Slutzki, G. (eds.) Grammatical Inference, 4th International Colloquium, ICGI-98, Ames, Iowa, USA, July 12-14, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1433, pp. 1–12. Springer (1998). https://doi.org/10.1007/BFB0054059
24. Lauffer, N., Yalcinkaya, B., Vazquez-Chanlatte, M., Shah, A., Seshia, S.A.: Learning deterministic finite automata decompositions from examples and demonstrations. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided

Design, FMCAD 2022, Trento, Italy, October 17-21, 2022, pp. 1–6. IEEE (2022). https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_39
25. Myhill, J.: Finite automata and the representation of events. In: Technical Report WADD TR-57-624, pp. 112–137 (1957)
26. Neider, D.: Computing minimal separating DFAs and regular invariants using SAT and SMT solvers. In: Chakraborty, S., Mukund, M. (eds.) Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7561, pp. 354–369. Springer (2012). https://doi.org/10.1007/978-3-642-33386-6_28
27. Neider, D., Jansen, N.: Regular model checking using solver technologies and automata learning. In: Brat, G., Rungta, N., Venet, A. (eds.) NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7871, pp. 16–31. Springer (2013). https://doi.org/10.1007/978-3-642-38088-4_2
28. Nerode, A.: Linear automaton transformations. Am. Math. Soc. **4** 541–544 (1958)
29. Oncina, J., Garcia, P.: Inferring regular languages in polynomial updated time. In: Pattern Recognition and Image Analysis: Selected Papers from the IVth Spanish Symposium, pp. 49–61. World Scientific (1992)
30. Pfleeger, C.P.: State reduction in incompletely specified finite-state machines. IEEE Trans. Computers **22**(12), 1099–1102 (1973). https://doi.org/10.1109/T-C.1973.223655
31. Pitt, L., Warmuth, M.K.: The minimum consistent DFA problem cannot be approximated within any polynomial. J. ACM **40**(1), 95–142 (1993). https://doi.org/10.1145/138027.138042
32. Smetsers, R., Fiterau-Brostean, P., Vaandrager, F.W.: Model learning as a satisfiability modulo theories problem. In: Klein, S.T., Martín-Vide, C., Shapira, D. (eds.) Language and Automata Theory and Applications - 12th International Conference, LATA 2018, Ramat Gan, Israel, April 9-11, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10792, pp. 182–194. Springer (2018). https://doi.org/10.1007/978-3-319-77313-1_14
33. Trakhtenbrot, B.A., Barzdin, Y.M.: Finite Automata: Behavior And Synthesis. Elsevier (1973)
34. Ulyantsev, V., Zakirzyanov, I., Shalyto, A.: BFS-based symmetry breaking predicates for DFA identification. In: Dediu, A., Formenti, E., Martín-Vide, C., Truthe, B. (eds.) Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings. Lecture Notes in Computer Science, vol. 8977, pp. 611–622. Springer (2015). https://doi.org/10.1007/978-3-319-15579-1_48
35. Verwer, S., Hammerschmidt, C.A.: flexfringe: a passive automaton learning package. In: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017, pp. 638–642. IEEE Computer Society (2017). https://doi.org/10.1109/ICSME.2017.58
36. Zakirzyanov, I., Morgado, A., Ignatiev, A., Ulyantsev, V., Marques-Silva, J.: Efficient symmetry breaking for sat-based minimum DFA inference. In: Martín-Vide, C., Okhotin, A., Shapira, D. (eds.) Language and Automata Theory and Applications - 13th International Conference, LATA 2019, St. Petersburg, Russia, March 26-29, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11417, pp. 159–173. Springer (2019). https://doi.org/10.1007/978-3-030-13435-8_12

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Visualizing Game-Based Certificates for Hyperproperty Verification

Raven Beutner(✉), Bernd Finkbeiner, and Angelina Göbl

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{raven.beutner,finkbeiner,angelina.goebl}@cispa.de

Abstract. Hyperproperties relate multiple executions of a system and are commonly used to specify security and information-flow policies. While many verification approaches for hyperproperties exist, providing a convincing *certificate* that the system satisfies a given property is still a major challenge. In this paper, we propose *strategies* as a suitable form of certificate for hyperproperties specified in a fragment of the temporal logic HyperLTL. Concretely, we interpret the verification of a HyperLTL property as a game between universal and existential quantification, allowing us to leverage strategies for the existential quantifiers as certificates. We present `HyGaViz`, a browser-based visualization tool that lets users interactively explore an (automatically synthesized) witness strategy by taking control over universally quantified executions.

1 Introduction

Hyperproperties [17] relate multiple execution traces of a system and occur frequently when reasoning about information flow [35,38], robustness [12,15], independence [3], knowledge [10,14], and causality [19,25]. A popular logic for specifying temporal hyperproperties is HyperLTL [16], an extension of LTL with explicit quantification over execution traces. For example, we can use HyperLTL to express a simple non-interference property as follows:

$$\forall \pi_1 . \exists \pi_2 . \square (l_{\pi_1} \leftrightarrow l_{\pi_2}) \land \square (o_{\pi_1} \leftrightarrow o_{\pi_2}) \land \square (\neg h_{\pi_2}) \qquad (\varphi_{NI})$$

Informally, this property – called non-inference [33] – requires that any possible observation made via the low-security input (modeled via atomic proposition l) and output (o) is compatible with a fixed "dummy" sequence of high-security inputs (h) [33]. Concretely, φ_{NI} states that for *any* execution π_1, *some* execution π_2 combines the low-security observations of π_1 with fixed dummy values for h; here, we require that h is constantly set to false, i.e., $\square(\neg h_{\pi_2})$ (cf. [23]).

Verification and Certificates. In recent years, many verification techniques for temporal hyperproperties (expressed, e.g., in HyperLTL) have been developed [2,8,9,16,24,30,35]. However, while *checking* if a given system satisfies a HyperLTL property is important, an often equally critical aspect is to convince the user

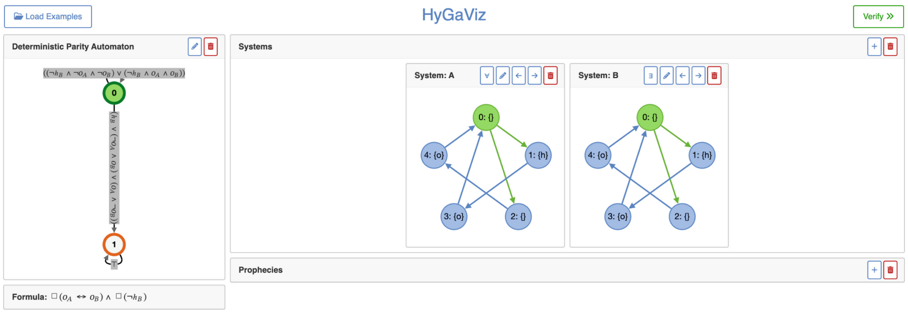
© The Author(s) 2025

of this satisfaction using explainable *certificates.* For trace properties – specified, e.g., in LTL – user-understandable certificates for positive and negative verification results have been explored extensively [4,5,13,27,28,32]. Likewise, for alternation-free HyperLTL formulas (i.e., formulas that use a single type of quantifier), known techniques for LTL apply [29]. In contrast, generating explainable certificates for the satisfaction of alternating properties like φ_{NI} is more complex. For example, φ_{NI} states that for any trace π_1, there exists some matching execution π_2. A certificate must thus implicitly define a mapping that, given a concrete choice for π_1, produces a witness trace π_2. Defining and understanding such a mapping can be complex, even for simple systems with few states.

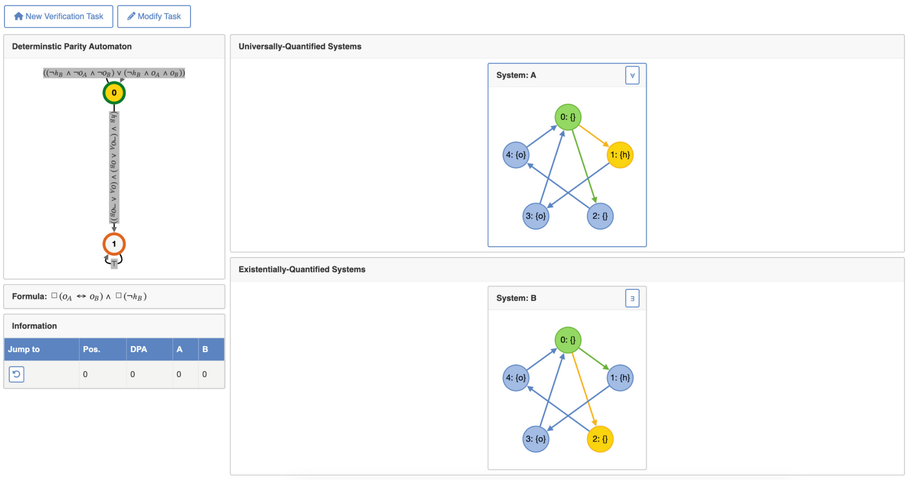
Strategies as Certificates. In this paper, we propose *strategies* as certificates for the satisfaction of $\forall^*\exists^*$ HyperLTL formulas (i.e., formulas where an arbitrary number of universal quantifiers is followed by an arbitrary number of existential quantifiers; e.g., φ_{NI}). To accomplish this, we take a game-based verification perspective [6,20]. The key idea is to interpret the verification of a $\forall\pi_1.\exists\pi_2.\,\psi$ formula (where ψ is the LTL body) as a game between universal and existential quantification. The $\forall$-player controls the universally quantified trace by moving through the system (thereby producing a trace π_1), and the $\exists$-player reacts with moves in a separate copy of the system (thereby producing a trace π_2). Any strategy for the $\exists$-player that ensures that π_1 and π_2, together, satisfy ψ, implies that the formula is satisfied on the given system. We can think of a winning strategy as a step-wise Skolem function that, for every trace π_1, iteratively constructs a witnessing trace π_2.

Visualizing Strategies. In this paper, we introduce `HyGaViz`, a verification and visualization tool for strategies in the context of HyperLTL verification. In `HyGaViz`, the user can input (possibly identical) finite-state transition systems and a HyperLTL formula φ. `HyGaViz` then automatically attempts to synthesize a strategy that witnesses the satisfaction of φ. If a strategy exists, `HyGaViz` displays it to the user. Our key insight is that we can let the user explore the strategy interactively by taking control of universally quantified traces. That is, instead of displaying the strategy in its entirety (e.g., as a table or decision diagram), we let the user play a game. In each step of the game, the user decides on a successor state for each universally quantified system (i.e., the user takes the role of the $\forall$-player), and `HyGaViz` automatically updates the states of all existentially quantified systems (i.e., `HyGaViz` plays the role of the $\exists$-player).

Example 1. We consider a simple verification instance in Fig. 1. On `HyGaViz`'s initial page (Fig. 1a), we create two (in this case, equal) transition systems (labeled A, B) over atomic propositions (APs) o and h, depicted in the top right. In each system, each state is identified by a natural number and lists all APs that hold in the given state. From initial state 0, the system can branch on AP h (states 1 and 2), but, in either case, AP o is set in the next step (states 3 and 4). We want to verify φ_{NI}, which – due to the absence of low-security input l – simplifies to $\forall A.\exists B.\square(o_A \leftrightarrow o_B) \wedge \square(\neg h_B)$. Note how, in `HyGaViz`, the


Load Examples
HyGaViz
Verify »
Deterministic Parity Automaton
$((\neg h_B \wedge \neg o_A \wedge \neg o_B) \vee (\neg h_B \wedge o_A \wedge o_B))$
0
1
$\top$
Systems
System: A
System: B
0: {}
1: {h}
2: {}
3: {o}
4: {o}
Prophecies
Formula: $\square(o_A \leftrightarrow o_B) \wedge \square(\neg h_B)$

(a)


New Verification Task
Modify Task
Deterministic Parity Automaton
$((\neg h_B \wedge \neg o_A \wedge \neg o_B) \vee (\neg h_B \wedge o_A \wedge o_B))$
0
1
$\top$
Universally-Quantified Systems
System: A
Existentially-Quantified Systems
System: B
0: {}
1: {h}
2: {}
3: {o}
4: {o}
Formula: $\square(o_A \leftrightarrow o_B) \wedge \square(\neg h_B)$
Information

Jump to	Pos.	DPA	A	B
↻	0	0	0	0

(b)

Fig. 1. Screenshots of `HyGaViz`. (Color figure online)

quantifier prefix is determined implicitly by the order and quantifier type of the systems, and the LTL body is displayed on the bottom left. The user can change the systems, the quantification type, the name, and the order of the systems using the buttons above each system. Upon entering the LTL formula, `HyGaViz` automatically displays a deterministic automaton for the property (top left). After clicking the *Verify* button (top right), the user is directed to the strategy simulation page (depicted in Fig. 1b). During the simulation, `HyGaViz` displays the current state of the automaton and the system state for A and B (in green) and lets the user control the state of (the universally quantified) system A. By hovering over the successor state of system A, `HyGaViz` highlights the next state for system B (in yellow). In this instance, systems A and B are both in state 0. When the user moves system A to state 1, `HyGaViz` reacts by moving system B

to state 2 (as it has to ensure $\square(\neg h_B)$). By clicking on a successor state for A, the user locks the choice, and the game progresses to the next round. $\triangle$

Related Work. `HyperVis` [29] is a tool for the visualization of counterexample traces for alternation-free $\forall^k$ formulas. Notably, a counterexample to a $\forall^k$ property is a concrete list of k traces, so visualization is possible by highlighting the relevant parts of the traces, potentially using causality-based techniques [18]. Our visualization for properties involving quantifier alternations is rooted in the game-based verification approach for HyperLTL [6, 20], which becomes complete when adding prophecies [6] (see Sect. 2.2). To the best of our knowledge, we are the first to propose a principled approach to generate and visualize user-understandable certificates for alternating hyperproperties.

2 HyperLTL, Game-Based Verification, and Prophecies

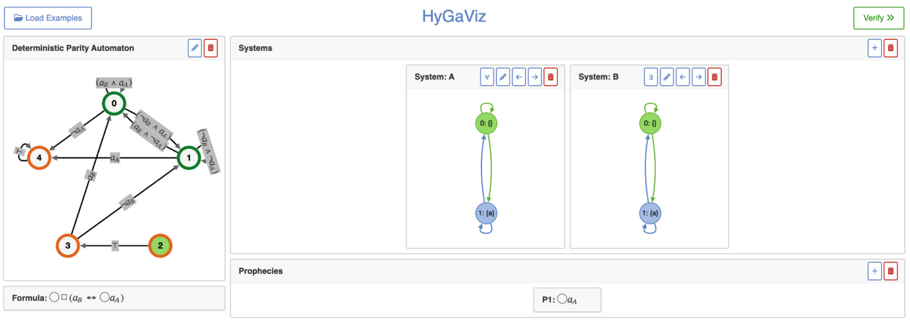
We fix a finite set of atomic propositions AP. A transition system (TS) is a tuple $\mathcal{T} = (S, s_{init}, \kappa, L)$, where S is a finite set of states, $s_{init} \in S$ is an initial state, $\kappa : S \to (2^S \setminus \{\emptyset\})$ is a transition function, and $L : S \to 2^{AP}$ is a state labeling. HyperLTL formulas are generate by the following grammar

$$\psi := a_\pi \mid \psi \wedge \psi \mid \neg\psi \mid \bigcirc\psi \mid \psi\,\mathcal{U}\,\psi \qquad \varphi := \forall\pi.\,\varphi \mid \exists\pi.\,\varphi \mid \psi$$

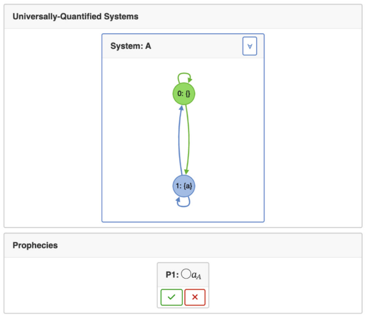
where $a \in AP$ is an atomic proposition, and π is a trace variable. In a HyperLTL formula, we can quantify over traces in a system (bound to some trace variable), and then evaluate an LTL formula on the resulting traces. In the LTL body, formula a_π expresses that AP a should hold in the current step on the trace bound to trace variable π. See [23] for details.

2.1 Game-Based Verification

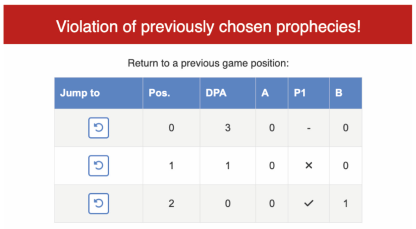
`HyGaViz`'s verification certificates are rooted in a game-based verification method [6]. Given a $\forall^*\exists^*$ HyperLTL formula $\forall\pi_1 \ldots \forall\pi_k.\, \exists\pi_{k+1} \ldots \exists\pi_{k+l}.\, \psi$, we view verification as a game between the $\forall$-player (controlling traces $\pi_1, \ldots, \pi_k$) and the $\exists$-player (controlling traces $\pi_{k+1} \ldots, \pi_{k+l}$). Each state of the game has the form $\langle s_1, \ldots, s_{k+l}, q\rangle$, where $s_1, \ldots, s_{k+l} \in S$ are system states (representing the current state of $\pi_1, \ldots, \pi_{k+l}$, respectively), and q is the state of a deterministic parity automaton (DPA) that tracks the acceptance of the LTL body ψ. When the game is in state $\langle s_1, \ldots, s_{k+l}, q\rangle$, the $\forall$-player first fixes successor states $s'_1, \ldots, s'_k$ for $\pi_1, \ldots, \pi_k$ (such that $s'_i \in \kappa(s_i)$ for all $1 \leq i \leq k$); the $\exists$-player responds by selecting successor states $s'_{k+1}, \ldots, s'_{k+l}$ for $\pi_{k+1}, \ldots, \pi_{k+l}$; and the game repeats from state $\langle s'_1, \ldots, s'_{k+l}, q'\rangle$ (where q' is the updated DPA state).



(a)



(b)



Violation of previously chosen prophecies!

Return to a previous game position:

Jump to	Pos.	DPA	A	P1	B
↺	0	3	0	-	0
↺	1	1	0	×	0
↺	2	0	0	✓	1

(c)

Fig. 2. Screenshots of `HyGaViz` when using prophecies.

Visualizing Game-Based Verification. In `HyGaViz`, the user can create a verification scenario by manually creating finite-state transition systems and a HyperLTL formula; see Fig. 1a. Note how the quantification prefix is determined implicitly by the order of the systems. In particular, the traces are resolved on individual (potentially different) transition systems. During simulation (cf. the example in Fig. 1b), we visualize a game state $\langle s_1, \dots, s_{k+l}, q\rangle$ by marking the current state of each system – separated into user-controlled (universally quantified) systems (top right) and strategy-controlled (existentially quantified) ones (bottom right) – and display the current state of the DPA (top left). The user takes the role of the ∀-player and, in each step, determines successor states for all universally quantified systems. Once successor states for all universally quantified systems are confirmed, `HyGaViz` automatically updates existentially quantified systems (and the DPA state) based on the internally computed strategy, and the game continues to the next stage. Moreover, `HyGaViz` highlights the next states when the user *hovers* over possible successor states for the universally quantified systems (once successor states for all but one universally quantified system are confirmed). Using the information tab in the bottom left, the user can jump to previous game states and explore the reaction of the strategy to different choices for the universally quantified systems.

2.2 Prophecies

In our game, the $\exists$-player only observes a finite prefix of the traces produced by the $\forall$-player (or, equivalently, the user of `HyGaViz`) and is thus missing information about the future. We can counteract this by using *prophecies* [1], which are LTL formulas over trace variables $\pi_1, \dots, \pi_k$ [6]. Given an LTL prophecy formula θ, the $\forall$-player (i.e., the user) has to, in each step, decide if its future behavior (on $\pi_1, \dots, \pi_k$) satisfies θ. If the $\forall$-player decides that θ holds (resp. does not hold), the $\exists$-player can play under the assumption that the *future* behavior of the $\forall$-player satisfies (resp. violates) θ. See [6] for details.

Example 2. We illustrate prophecies with the example in Fig. 2. The two systems A and B in Fig. 2a generate all traces over AP a, and the HyperLTL formula $\forall A. \exists B. \bigcirc \Box (a_B \leftrightarrow \bigcirc a_A)$ requires that trace B *predicts* the future behavior of A. Without prophecies, the $\exists$-player loses: No matter what successor state the $\exists$-player picks, the $\forall$-player can, in the next step, violate the prediction of the $\exists$-player. `HyGaViz` communicates the absence of a winning strategy if the user pushes the *Verify* button. Instead, the user can add the LTL prophecy $\bigcirc a_A$ (cf. Fig. 2a). During simulation, the user (who takes the role of the $\forall$-player) has to, in each step, fix a successor state for system A *and* determine if prophecy $\bigcirc a_A$ holds. We depict an excerpt of the simulation page in Fig. 2b. As expected, the strategy for the $\exists$-player (computed automatically by `HyGaViz`) can use the prophecy to win: For example, if the user states that $\bigcirc a_A$ holds (so the $\exists$-player can assume that a hold in the next step in A), `HyGaViz` moves system B to state 1. If the user violates a previous prophecy decision – e.g., by stating that prophecy $\bigcirc a_A$ holds but, in the next step, moving system A to state 0 where AP a does not hold – `HyGaViz` detects this violation and forces the user to restart from an earlier state of the game (Fig. 2c). $\triangle$

3 HyGaViz: Tool Overview

`HyGaViz` consists of a backend verification engine written in `F#`. The backend uses `spot` [22] to translate LTL formulas to DPAs and `oink` [21] to synthesize a strategy for the $\exists$-player. We use a stateless `Node.js` [37] backend that communicates with the verification engine via `JSON`. `HyGaViz`'s frontend is written in `JavaScript` and uses `Cytoscape.js` [26] to render transition systems and automata.

4 Conclusion

We have proposed the first method to generate and visualize certificates for the satisfaction of $\forall^*\exists^*$ HyperLTL formulas. Our tool, `HyGaViz`, allows users to *interactively* explore the complex dependencies between multiple traces by challenging a strategy for existentially quantified traces. Ultimately, `HyGaViz` is a first step to foster trust in (and understanding of) verification results for complex alternating hyperproperties, as is needed to, e.g., certify information-flow policies like φ_{NI}. For now, `HyGaViz` can handle (small) finite state systems, which we visualize as directed graphs. The underlying strategy-centered approach also applies to larger (potentially infinite-state) systems represented symbolically [7]. In future work, one could extend `HyGaViz` to such systems by exploring different visualization approaches for larger systems [31,34,36].

Acknowledgments. This work was partially supported by the European Research Council (ERC) Grant HYPER (101055412) and by the German Research Foundation (DFG) as part of TRR 248 (389792660).

Data Availability Statement. `HyGaViz` is available at [11].

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2), 253–284 (1991). https://doi.org/10.1016/0304-3975(91)90224-P
2. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. (2011). https://doi.org/10.1017/S0960129511000193
3. Bartocci, E., Henzinger, T.A., Nickovic, D., da Costa, A.O.: Hypernode automata. In: International Conference on Concurrency Theory, CONCUR 2023 (2023). https://doi.org/10.4230/LIPICS.CONCUR.2023.21
4. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.J.: Explaining counterexamples using causality. In: International Conference on Computer Aided Verification, CAV 2009 (2009). https://doi.org/10.1007/978-3-642-02658-4_11
5. Beschastnikh, I., Liu, P., Xing, A., Wang, P., Brun, Y., Ernst, M.D.: Visualizing distributed system executions. ACM Trans. Softw. Eng. Methodol. (2020). https://doi.org/10.1145/3375633
6. Beutner, R., Finkbeiner, B.: Prophecy variables for hyperproperty verification. In: Computer Security Foundations Symposium, CSF 2022 (2022). https://doi.org/10.1109/CSF54842.2022.9919658
7. Beutner, R., Finkbeiner, B.: Software verification of hyperproperties beyond k-safety. In: International Conference on Computer Aided Verification, CAV 2022 (2022). https://doi.org/10.1007/978-3-031-13185-1_17
8. Beutner, R., Finkbeiner, B.: AutoHyper: explicit-state model checking for HyperLTL. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023 (2023). https://doi.org/10.1007/978-3-031-30823-9_8
9. Beutner, R., Finkbeiner, B.: Non-deterministic planning for hyperproperty verification. In: International Conference on Automated Planning and Scheduling, ICAPS 2024 (2024). https://doi.org/10.1609/ICAPS.V34I1.31457
10. Beutner, R., Finkbeiner, B., Frenkel, H., Metzger, N.: Second-order hyperproperties. In: International Conference on Computer Aided Verification, CAV 2023 (2023). https://doi.org/10.1007/978-3-031-37703-7_15

11. Beutner, R., Finkbeiner, B., Göbl, A.: HyGaViz: visualizing game-based certificates for hyperproperty verification (2024). https://doi.org/10.5281/zenodo.12206584
12. Biewer, S., et al.: Conformance relations and hyperproperties for doping detection in time and space. Log. Methods Comput. Sci. **18**, 14 (2022). https://doi.org/10.46298/lmcs-18(1:14)2022 https://doi.org/10.46298/lmcs-18(1:14)2022
13. Bolton, M.L., Bass, E.J.: Using task analytic models to visualize model checker counterexamples. In: International Conference on Systems, Man and Cybernetics, SMC 2010 (2010). https://doi.org/10.1109/ICSMC.2010.5641711
14. Bozzelli, L., Maubert, B., Pinchinat, S.: Unifying hyper and epistemic temporal logics. In: International Conference on Foundations of Software Science and Computation Structures, FoSSaCS 2015 (2015). https://doi.org/10.1007/978-3-662-46678-0_11
15. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity and robustness of programs. Commun. ACM (2012). https://doi.org/10.1145/2240236.2240262
16. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: International Conference om Principles of Security and Trust, POST 2014 (2014). https://doi.org/10.1007/978-3-642-54792-8_15
17. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. (2010). https://doi.org/10.3233/JCS-2009-0393
18. Coenen, N., et al.: Explaining hyperproperty violations. In: International Conference on Computer Aided Verification, CAV 2022 (2022). https://doi.org/10.1007/978-3-031-13185-1_20
19. Coenen, N., Finkbeiner, B., Frenkel, H., Hahn, C., Metzger, N., Siber, J.: Temporal causality in reactive systems. In: International Symposium on Automated Technology for Verification and Analysis, ATVA 2022 (2022). https://doi.org/10.1007/978-3-031-19992-9_13
20. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: International Conference on Computer Aided Verification, CAV 2019 (2019). https://doi.org/10.1007/978-3-030-25540-4_7
21. van Dijk, T.: Oink: an implementation and evaluation of modern parity game solvers. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2018 (2018). https://doi.org/10.1007/978-3-319-89960-2_16
22. Duret-Lutz, A., et al.: From spot 2.0 to spot 2.10: what's new? In: International Conference on Computer Aided Verification, CAV 2022 (2022). https://doi.org/10.1007/978-3-031-13188-2_9
23. Finkbeiner, B.: Logics and algorithms for hyperproperties. ACM SIGLOG News (2023). https://doi.org/10.1145/3610392.3610394
24. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: International Conference on Computer Aided Verification, CAV 2015 (2015). https://doi.org/10.1007/978-3-319-21690-4_3
25. Finkbeiner, B., Siber, J.: Counterfactuals modulo temporal logics. In: International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2023 (2023). https://doi.org/10.29007/QTW7
26. Franz, M., Lopes, C.T., Huck, G., Dong, Y., Sümer, S.O., Bader, G.D.: Cytoscape.js: a graph theory library for visualisation and analysis. Bioinformatics **32**, 309–311 (2016). https://doi.org/10.1093/BIOINFORMATICS/BTV557
27. Griggio, A., Roveri, M., Tonetta, S.: Certifying proofs for LTL model checking. In: Formal Methods in Computer Aided Design, FMCAD 2018 (2018). https://doi.org/10.23919/FMCAD.2018.8603022

28. Groce, A., Kroening, D., Lerda, F.: Understanding counterexamples with explain. In: International Conference on Computer Aided Verification, CAV 2004 (2004). https://doi.org/10.1007/978-3-540-27813-9_35
29. Horak, T., et al.: Visual analysis of hyperproperties for understanding model checking results. IEEE Trans. Vis. Comput. Graph. (2022). https://doi.org/10.1109/TVCG.2021.3114866
30. Hsu, T., Sánchez, C., Bonakdarpour, B.: Bounded model checking for hyperproperties. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2021 (2021). https://doi.org/10.1007/978-3-030-72016-2_6
31. Jerding, D.F., Stasko, J.T., Ball, T.: Visualizing interactions in program executions. In: International Conference on Software Engineering, ICSE 1997 (1997). https://doi.org/10.1145/253228.253356
32. Kasenberg, D., Thielstrom, R., Scheutz, M.: Generating explanations for temporal logic planner decisions. In: International Conference on Automated Planning and Scheduling, ICAPS 2020 (2020). https://doi.org/10.1609/icaps.v30i1.6740
33. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: Symposium on Security and Privacy, SP 1994 (1994). https://doi.org/10.1109/RISP.1994.296590
34. Moreno, A., Myller, N., Sutinen, E., Ben-Ari, M.: Visualizing programs with Jeliot 3. In: Conference on Advanced Visual Interfaces, AVI 2004 (2004). https://doi.org/10.1145/989863.989928
35. Rabe, M.N.: A temporal logic approach to information-flow control. Ph. D. thesis, Saarland University (2016)
36. Rajala, T., Laakso, M., Kaila, E., Salakoski, T.: Effectiveness of program visualization: a case study with the ViLLE tool. J. Inf. Technol. Educ. Innov. Pract. **7**, 15 (2008)
37. Tilkov, S., Vinoski, S.: Node.js: using javascript to build high-performance network programs. IEEE Internet Comput. **14**, 80–83 (2010). https://doi.org/10.1109/MIC.2010.145
38. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Computer Security Foundations Workshop CSFW 2003 (2003). https://doi.org/10.1109/CSFW.2003.1212703

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Chamelon : A Delta-Debugger for OCaml

Milla Valnet[1,2,3](✉), Nathanaëlle Courant[3], Guillaume Bury[3], Pierre Chambart[3], and Vincent Laviron[3]

[1] École Normale Supérieure, Université PSL, 75005 Paris, France
milla.valnet@lip6.fr
[2] Sorbonne Université, CNRS, LIP6, 75005 Paris, France
[3] OCamlPro, 75014 Paris, France

Abstract. Tools that manipulate OCaml code can sometimes fail even on correct programs. Identifying and understanding the cause of the error usually involves manually reducing the size of the program, so as to obtain a shorter program causing the same error—a long, sometimes complex and rarely interesting task. Our work consists in automating this task using a minimiser, or *delta-debugger*. To do so, we propose a list of unitary heuristics, i.e. small-scale reductions, applied through a dichotomy-based state-of-the-art algorithm. These proposals are implemented in the free Chamelon tool. Although designed to assist the development of an OCaml compiler, Chamelon can be adapted to all kinds of projects that manipulate OCaml code. It can analyse multifile projects and efficiently minimise real-world programs, reducing their size by one to several orders of magnitude. It is currently used to assist the industrial development of the flambda2 optimising compiler.

1 Introduction

Program errors sometimes occur oten large inputs, of hundreds or even thousands of lines. Identifying and isolating the error is often a long and tedious task, which generally involves manually minimising the size of the input as much as possible. The aim of a minimiser is to automate this work.

Sometimes called delta-debugging, this idea was developed in 1999 by Andreas Zeller [11] in order to isolate the cause of a program error by iteratively applying simplifications. It is defined as a methodology reducing a problem while preserving a certain property—here, the error. The tool thus does not eliminate the error, but on the contrary points to it.

This method is already used for languages such as C, with C-reduce [1], SMT-lib [6], or via implementations of Zeller's original work [11]. Nonetheless, the problem remains well studied. Zeller worked with Hildebrandt [10] to identify the inputs and interactions that cause programs' failure, using Mozilla browser user inputs as case study, and then demonstrated with Cleve [3] that delta-debugging works just as well for identifying errors due to the code itself as to its parameters. Seeing any debugging tasks as special cases of minimisation problems, he uses this method with Choi [2] for thread scheduling failures, and

© The Author(s) 2025

with Cleve [4] to identify which variables and at which execution step the error occurs. Finally, Leitner et al. combine this approach with slicing to reduce the size of failure cases in random test generation. Some also improved the state of the art with machine learning [5], probabilistic algorithms [9], etc.

In OCaml, however, the existing debugging tools are limited to type errors [7]. This project therefore proposes the first general-purpose minimiser for OCaml code, Chamelon. Although initially designed to assist an OCaml compiler development in the industry, such a tool may prove useful for other projects using or manipulating OCaml code. This work makes the following contributions:

- a list of OCaml-specific minimisation heuristics;
- combined with a state-of-the-art technique to perform dichotomy-based minimisations;
- an OCaml implementation supporting multi-file projects and runtime errors available as open-source software;
- with a modular design to support the development of various kinds of OCaml projects.

Outline. Sect. 2 presents the tool usage. Section 3 explains the unitary heuristics proposed to minimise the program, while Sect. 4 explains how they are combined. Section 5 shows extensions of this work.

2 Tool Usage

2.1 Development Context

The tool Chamelon is a delta-debugger for OCaml programs, available as open-source software on GitHub[1]. Earlier results on Chamelon were presented in French [8]. It was originally designed to support the development of the flambda2 optimising compiler[2], developed by OCamlPro and used in particular by Jane Street. Indeed, when flambda2 failed on programs correct according to the standard compiler, identifying the error cause in flambda2 was not always easy. However, Chamelon is built in a modular way, reducing a program size while ensuring an user-given condition, and can be used in various context.

2.2 Usage

```
chamelon input -c command -e error
```

To use `chamelon`, all we need to do is giving it an `input` file, a `command` to execute and an `error`, that is, the string we want to find in the command's standard output. `chamelon` then prints a log of applied transformations in the standard output, and when done, the output is a minimised version of the input,

[1] https://github.com/Ekdohibs/chamelon.
[2] https://github.com/ocaml-flambda/flambda-backend.

such that the command output still contains the error. To minimise a set of files, we only need to provide the command with several inputs. This way, Chamelon can be used in different settings.

A simple real-world use case is available online[3]. By following instructions in `README-CHAMELON.md`, we can make Chamelon reduce the size of an input file trigerring a `Fatal_error` in flambda2, to help understand the origin of the error.

2.3 Experimental Results

The tool is currently used daily at OCamlPro to help flambda2 development for almost a year. It gave results on real cases of failure, significantly reducing the output program size. Among the experimental results, it was able to minimise a 650-lines program[4] that failed to compile into a program of just 6 lines causing the same error, identifying a problem in the optimisation of pattern matching[5]:

```
let offset ~byte_order  byte_n =
  match byte_order with | `Little_endian -> 0 | `Big_endian -> byte_n
let pack_unsigned_16 ~byte_order  =
  __ignore__ ((offset ) ~byte_order 0);
  __ignore__ ((__dummy__ ()) ((offset ) ~byte_order 1));
  __dummy__ ()
```

We also tested the minimiser on larger programs. For example, given a 3842-lines program on which the compiler was failing, the minimiser reduced it to 22 lines, in around thirty minutes on an average laptop. Often, the output can still be minimised by hand. However, the tool automates a large part of the work. Finally, in a multi-file framework, the minimiser is also able to merge or delete files, resulting in a minimised copy of the project that triggered the error.

Note that reducing the size of the program is not the only interesting action of the minimiser. Indeed, when a simplification is not done, it means that it removed the error, which can therefore be exploited. We not only benefit from the size-reducing, but also from the *minimality* of the program with regard to the heuristics.

3 Heuristics

The concept of the approach is to compose and combine different unitary heuristics, applying each of them as much as possible before trying the next. We present here the different heuristics implemented to minimise an OCaml program. It should be noted that, having initially targeted compilation problems, our approach aims much more at identifying errors caused by a certain code structure than by a certain semantics or execution: this therefore guides our choice of heuristics.

[3] https://github.com/Ekdohibs/flambda-backend/tree/chamelon-demo.

[4] https://github.com/janestreet/core_kernel/blob/master/binary_packing/src/binary_packing.ml.

[5] fixed by https://github.com/ocaml-flambda/flambda-backend/pull/1073.

3.1 Suppress Definitions

Delete definitions starting from the end. The first simple heuristics consists in deleting all definitions—of variables, types, modules, etc.—starting from the end. It aims at removing the code located after error's cause, on which the error does not depend.

Replace expressions by dummy values. When definitions cannot simply be removed, we try to replace them with the simplest possible values. The challenge is then to determine which trivial value we want to replace our expression with while respecting type constraint. For ground types, we simply replace expressions of type `int` by 0, those of type `float` by 0.0, those of type `char` by `'0'`, those of type `string` by `""` and those of type `unit` by `()`. For the other types, we used:

```
external __dummy__ : unit -> 'a = "%opaque"
```

Here, `__dummy__ ()` is of type `'a`, and can therefore replace an expression of any type. It is based on the external primitive `opaque`: when compiled, it is considered as a function returning an arbitrary value—here, a function of type `unit -> 'a` because of the annotation. However, at runtime, it behaves like the identity function: for this reason, the value of `__dummy__()` is `()`, causing a type error. When targetting compilation failures, this is not a limitation. However, to generalize the tool's use cases, this problem will be adressed in Sect. 5.

3.2 Simplify Abstract Data Types

Suppress constructors from ADTs. A first heuristic consists in deleting a constructor `Cons` from an algebraic data type. This involves propagating this deletion of in the code: expressions `Cons`$(e_1, \ldots, e_n)$ are replaced by `__dummy__ ()`, and patterns using `Cons` are simply removed.

Delete fields from record types or constructors. When deleting an entire constructor is not possible, we instead delete its fields. After deleting its ith field's definition, we go through the code to delete the ith field in `Cons(e1,..,en)` expressions, and the ith sub-pattern in each `Cons(p1,..,pn)` pattern—replacing variables bound by `pi` with `__dummy__ ()`.

3.3 Simplify Code

Modify attributes. We remove attributes of functions, modules, etc. from the program to make it less verbose. However, `local [never|always]` and `inline [never|always]` to functions can also provide valuable information about the origin of the failure, forcing the compiler's inlining strategies.

Inline functions. Inlining a function, i.e. replacing it with its definition at call site, can lead to additional simplifications.

Flatten modules. Flattening modules means removing variables definitions from `module Name = struct ... end` block. To avoid name conflicts between variables from the module and variables defined in the program, we chose to precede the name of the variable by the name of the origin module : this change is then propagated throughout the program.

3.4 Remove Simplification Artifacts

Situations that would not or only rarely appear in real user code may appear after applying the above heuristics:

Remove dead code. For each variable, module and type, we go, and when not used, we simply delete their definition.

Simplify pattern matching. When the match contains a unique one-variable pattern, we replace `match e1 with x -> e2` by `e2` in which `x` has been textually substituted by `e1`.

Sequentialize function calls. After simplifications, we may obtain a function application of the form `(__dummy__ ()) e1 ... en`. We sequentialise its by evaluating each argument separately, to get non-nested expressions. We use the primitive `external __ignore__ : 'a -> unit = "%ignore"` . We then transform `(__dummy__ ()) e1 ... en` into:
`__ignore__ e1 ; ... ; __ignore__ en ; __dummy__ ()`

Simplify `rec` and unused arguments. After replacing expressions and definitions by `dummmy`, arguments of a function may no longer be used. We then delete them and propagate their deletion to all of the function's call sites. When the ith argument of the function `f` is deleted, all occurrences of `f` are replaced by `(fun x1 ... xn -> f x1 ... xi-1 xi+1 ... xn)`. Similarly, when the function is no longer recursive, we remove the keyword `rec`.

Simplify sequences. Expressions of the form `(); e` are replaced by `e`.

4 The Iteration

A unitary heuristic can possibly be applied at different points in a program: when trying to delete a constructor from an ADT, many constructors are possible candidates. We call "n-th program point" the n-th position, while reading the program's AST, where it can be performed. When trying to apply it at a program point—e.g. deleting one of those constructors, there are three possible cases:

- This simplification does not remove the error: the program has been reduced!
- This simplification removes the error: we do not want to apply it.
- The index of the point is greater than that of the last modifiable point.

We iterate this way: we take as input the program, a heuristic, and a position. We then attempt to apply the heuristics at this position. If minimisation is possible, we iterate over the new program without incrementing the position, since after simplifying the nth point, the next modifiable point is the new nth. If minimisation is not possible, the next position is examined. Finally, if the position is too large, the whole program was examined, so we return.

Dichotomic Optimisation. In Chamelon, this loop is otpimized by dichotomy, as initially suggested by Keller [11], by no longer trying to minimise locations one by one, but rather a set of locations of length 2^n. This method improves efficiency by a factor of 10 on real programs of a few thousand lines.

Heuristics Order. The application order of the different heuristics was determined experimentally, on a small sample of tests, mainly by finishing with the heuristics removing the simplification artefacts. For more robust and efficient scheduling, further research and testing could prove useful.

5 Extensions

Multifiles. In real use cases, a project is made of multiple interdependant files. We have therefore adapted Chamelon to work on such projects:

- First, we try deleting as many files as possible, in the order of dependencies;
- Then, we try merging as much files as possible;
- Finally, each remaining file is minimised with previous methods.

Note that every object modification must be propagated to all dependencies. For example, if an argument of a function `f` is deleted, it must be deleted at each `f` call sites, in each of the program's dependencies. To use Chamelon in multifile mode, we need to provide it with the list of files to minimise, in dependencies order—which can be given by `ocamldep` tool.

Runtime. The work presented so far focused on compile-time errors. However, errors may also occur at runtime. To handle this, we replaced the `__dummy__` values, causing runtime errors, using an algorithm which, given an input type, generates an expression of the same type, as concise as possible.

Compatibility. The implementation uses OCaml compiler libraries to manipulate abstract syntax trees. A compatibility library is implemented, so that changing of compiler version only requires some information about the new AST.

Adding Heuristics. Implementing a new heuristics is low-cost: we only need to write the transformation through existing mappers function for OCaml AST.

6 Conclusion

In the future, an interesting extension would be to make the Chamelon minimiser compatible with `dune`—the OCaml build system. Finally, through its use in real-world examples, we aim at improving existing heuristics and finding new ones, so as to make it more robust, more efficient and faster. In the end, this work combines various minimisation heuristics with a state-of-the-art iteration technique and a modular design, offering the first delta-debugger for and in OCaml, available for its community!

Artifact. The artifact associated to this paper and demonstrating the use of Chamelon on different programs is available at https://doi.org/10.5281/zenodo.12520654.

References

1. C-reduce project. https://github.com/csmith-project/creduce
2. Choi, J.D., Zeller, A.: Isolating failure-inducing thread schedules. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 210–220 (2002)
3. Cleve, H., Zeller, A.: Finding failure causes through automated testing. In: Ducassé, M. (ed.) Proceedings of the Fourth International Workshop on Automated Debugging, AADEBUG 2000, Munich, Germany, 28–30 August 2000 (2000). https://arxiv.org/abs/cs/0012009
4. Cleve, H., Zeller, A.: Locating causes of program failures. In: Proceedings of the 27th International Conference on Software Engineering, pp. 342–351 (2005)
5. Heo, K., Lee, W., Pashakhanloo, P., Naik, M.: Effective program debloating via reinforcement learning. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, pp. 380–394. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3243734.3243838
6. Kremer, G., Niemetz, A., Preiner, M.: ddSMT 2.0: better delta debugging for the SMT-LIBv2 Language and friends. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 231–242. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_11
7. Sharrad, J., Chitil, O.: Refining the delta debugging of type errors. In: Proceedings of the 33rd Symposium on Implementation and Application of Functional Languages, IFL 2021, pp. 10–19. Association for Computing Machinery, New York (2022). https://doi.org/10.1145/3544885.3544888
8. Valnet, M., Courant, N., Bury, G., Chambart, P., Laviron, V.: Chamelon: un minimiseur pour et en ocaml. In: 35es Journées Francophones des Langages Applicatifs (JFLA 2024) (2024)
9. Wang, G., Shen, R., Chen, J., Xiong, Y., Zhang, L.: Probabilistic delta debugging. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, pp. 881–892. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3468264.3468625
10. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. **28**(2), 183–200 (2002). https://doi.org/10.1109/32.988498
11. Zeller, A.: Yesterday, my program worked. today, it does not. why? SIGSOFT Softw. Eng. Notes **24**(6), 253–267 (1999). https://doi.org/10.1145/318774.318946

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Automated Static Analysis of Quality of Service Properties of Communicating Systems

Carlos G. Lopez Pombo[1](✉), Agustín Eloy Martinez Suñé[2](✉), and Emilio Tuosto[3]

[1] Centro Interdisciplinario de Telecomunicaciones, Electrónica, Computación y Ciencia Aplicada, Universidad Nacional de Río Negro - Sede Andina and CONICET, San Carlos de Bariloche, Argentina
cglopezpombo@unrn.edu.ar

[2] CONICET–UBA. Instituto de Investigación en Ciencias de la Computación, Buenos Aires, Argentina
aemartinez@dc.uba.ar

[3] Gran Sasso Science Institute, L'Aquila, Italy
emilio.tuosto@gssi.it

Abstract. We present MoCheQoS, a bounded model checker to statically analyse Quality of Service (QoS) properties of message-passing systems. We consider QoS properties on measurable application-level attributes as well as resource consumption metrics, for example, those relating monetary cost to memory usage. The applicability of MoCheQoS is evaluated through case studies and experiments. A first case study is based on the AWS cloud while a second one analyses a communicating system automatically extracted from code. Additionally, we consider synthetically generated experiments to assess the scalability of MoCheQoS. These experiments showed that our model can faithfully capture and effectively analyse QoS properties in industrial-strength scenarios.

1 Introduction

Monolithic applications are steadily giving way to distributed cooperating components implemented as services. This transition was accelerated by the

Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233, the PRIN PNRR project DeLICE (F53D23009130001), "by the MUR dipartimento di eccellenza", by PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA - Advanced Space Technologies and Research Alliance , and Principles of Intelligent Behavior in Biological and Social Systems – PIBBSS, https://pibbss.ai/.
The authors thank the anonymous reviewers for their constructive comments and Omar Inverso for his suggestions.
Carlos G. Lopez Pombo—On leave from Instituto de Ciencias de la computación CONICET–UBA and Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires.

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 84–103, 2025.
https://doi.org/10.1007/978-3-031-71177-0_7

software-as-a-service motto triggered in the 21st century by the service-oriented computing (SOC) paradigm, later evolved in e.g., cloud, fog, or edge computing. These paradigms envisage software systems as applications running over globally available computational and networking infrastructures to procure services that can be composed on the fly so that, collectively, they can fulfil given goals [1].

Key to this trend are *Service Level Agreements* (SLAs) that express the terms and conditions under which services interact. An essential element covered by SLAs are the quantitative constraints specifying non-functional behaviour of services. For example, the current SLA and pricing scheme for the AWS Lambda service [2,3] declare constraints on quantifiable attributes. To the best of our knowledge, the standard practice is to informally specify the SLA of each service provided and then use run-time verification (like monitoring) to check quantitative non-functional properties. This approach makes it difficult to check system-level properties because SLAs (besides being informal) do not specify conditions on the composition of services.

Since their introduction in [4], choreographies stood out for a neat separation of concerns: choreographic models abstract away local computations focusing on the communications among participants; therefore, they are spot on for services since they reconcile the 'atomistic' view at the services' interactions level with the 'holistic' view at the system level. Indeed, choreographies require to specify a high-level description of interactions (the *global view*) and relate it to a description of services' behaviour (the *local view*). These are distinctive features of the choreographic framework presented in [5] to provide reasoning capabilities about the QoS of communicating systems, starting from the QoS of the underlying services. The basic idea in [5] is: (i) to specify constraints on quality attributes of local states of services and (ii) to verify through a bounded model-checking algorithm system-level QoS properties expressed in $\mathcal{QL}$, a specific dynamic logic where temporal modalities are indexed with *global choreographies* (g-choreographies [6], a formal model of global views of choreographies). A simple example can illustrate this. Let A be a service that converts files to various formats and invokes a storage service B to save the results of requests; both A and B charge customers depending on the size of stored data (as done e.g. by Amazon's DynamoDB service). The request of A to B can be abstracted away with two finite-state machines whose states are decorated with constraints on the two quality attributes: monetary cost (c) and data size (s) as follows:

$$\left\{\begin{matrix}\mathsf{c} \leqslant 5,\\ \mathsf{s} = 0\end{matrix}\right\} \; q_0 \xrightarrow{\mathsf{A}\,\mathsf{B}!\mathsf{s}} q_1 \; \left\{\begin{matrix}5 \leqslant \mathsf{c} \leqslant 10,\\ \mathsf{s} < 3\end{matrix}\right\} \quad \text{and} \quad \left\{\begin{matrix}\mathsf{c} = 0,\\ \mathsf{s} = 0\end{matrix}\right\} \; {q_0}' \xrightarrow{\mathsf{A}\,\mathsf{B}?\mathsf{s}} q_1' \; \left\{\begin{matrix}10 \leqslant \mathsf{s} \leqslant 50,\\ \mathsf{c} = 0.01 \cdot \mathsf{s}\end{matrix}\right\}$$

Intuitively, the formulae associated to states constraint the quality attributes upon the local computation executed in the states. For instance, both services store no data in their initial state; computation in A may cost up to five units before the request to B, which has no cost ($\mathsf{c} = 0$) in q_0' since it is just waiting to execute the input. If, as we assume, communication is asynchronous, then the composition of A and B yields a run like $\pi : s_0 \xrightarrow{\mathsf{A}\,\mathsf{B}!\mathsf{s}} s_1 \xrightarrow{\mathsf{A}\,\mathsf{B}?\mathsf{s}} s_2$ where first A sends the message and then B received it. Then, the system-level QoS of the composition of A and B would be the result of *aggregating* the constraints on c and s along the run π.

Structure & Contributions A main contribution of this paper is a tool to support the static analysis technique of QoS properties of message-passing systems. More precisely, we implement the bounded model-checking algorithm introduced in [5] (and summarised in Sect. 2) in a tool called MoCheQoS (after Model-Checker for QoS properties). By combining the SMT solver Z3 [7] and the choreographic development toolchain ChorGram [8–10] (as discussed in Sect. 3), MoCheQoS can model-check QoS properties expressed in $\mathcal{QL}$, the dynamic temporal logic of [5]. MoCheQoS is publicly available at [11].

A key feature of our approach is that the analysis of QoS properties of systems builds on the QoS constraints specified on the components of the system; as seen in the example above, MoCheQoS features the capability of aggregating QoS constraints along the computation of systems.

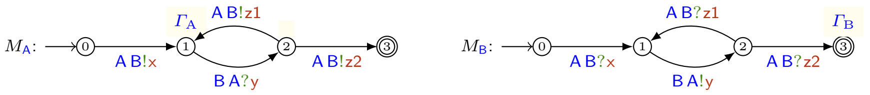
Another contribution is the empirical evaluation of our approach (Sect. 4), which is done through: (*a*) a case study borrowed from the AWS Cloud [12], (*b*) a case study borrowed from the literature [13] where communication protocols are automatically extracted from code, and (*c*) synthetic examples designed to evaluate the scalability of MoCheQoS.

Section 5 discusses related work; Sect. 6 concludes and sketches future work.

2 Preliminaries

We fix a set $\mathcal{P}$ of *participants* (identifying interacting services) and a set $\mathcal{M}$ of (types of) *messages* such that $\mathcal{P} \cap \mathcal{M} = \varnothing$. The communication model of MoCheQoS hinges on *QoS-extended communicating finite-state machines* [5] (qCFSMs for short). A CFSM [14] is a finite state automaton whose transitions are labelled by output or input actions. An output action A B!m (resp. input action A B?m) specifies the output (resp. input) of a message m from A to B (resp. received by B from A). A qCFSM is a CFSM where *QoS specifications*, that is first-order formulae predicating over QoS attributes, decorate states. (Unlike CFSMs, qCFSMs feature *accepting* states, represented here as double circles.)

Example 1. Let $\Gamma_{\mathrm{A}} = \{5 \leqslant \mathsf{mem} \leqslant 10, \mathsf{cost} = 0.2 \cdot \mathsf{mem}\}$ and $\Gamma_{\mathrm{B}} = \{\mathsf{mem} = 0, \mathsf{cost} = 1\}$ be two QoS specifications. In the system made of the qCFSMs



participant A first sends message x to B, then B and A exchange messages y and z1 an unbounded number of times, and finally A sends message z2 to B. ◇

A *QoS-extended communicating system* (qCS for short) is a map assigning a qCFSM to participants in $\mathcal{P}$; for instance, the map S where $S(\mathsf{A}) = M_{\mathsf{A}}$ and $S(\mathsf{B}) = M_{\mathsf{B}}$ are the qCFSM of Example 1 is a communicating system. Since QoS specifications do not affect communications, the semantics of qCSs is as the one of communicating systems. Let us recall how CFSMs interact.

Communicating systems are asynchronous: the execution of an output action $\mathsf{A}\,\mathsf{B}!\mathsf{m}$ allows the sender A to continue even if the receiver B is not ready to receive; message m is appended in an infinite FIFO buffer, the *channel* $\mathsf{A}\,\mathsf{B}$, from where B can consume m. Formally, given a communicating system S on $\mathcal{P}$, we define a labelled transition system (LTS) whose transitions relate *configurations* and communication actions. A configuration is a pair $\langle \mathfrak{q} \ ; \ \mathfrak{b} \rangle$ where $\mathfrak{q}$ and $\mathfrak{b}$ respectively maps each participant A to a state of $S(\mathsf{A})$ and each channel to a sequence of messages; state $\mathfrak{q}(\mathsf{A})$ keeps track of the state of machine $S(\mathsf{A})$ and buffer $\mathfrak{b}(\mathsf{A}\,\mathsf{B})$ yields the messages sent from A to B and not yet consumed. Let s_0 denote the *initial* configuration where, for all $\mathsf{A} \in \mathcal{P}$, $\mathfrak{q}(\mathsf{A})$ is the initial state of $S(\mathsf{A})$ and $\mathfrak{b}(\mathsf{A}\,\mathsf{B})$ is the empty sequence for all channels $\mathsf{A}\,\mathsf{B}$.

A configuration $\langle \mathfrak{q} \ ; \ \mathfrak{b} \rangle$ *reaches* another configuration $\langle \mathfrak{q}' \ ; \ \mathfrak{b}' \rangle$ *with a transition* ℓ if there is a message $\mathsf{m} \in \mathcal{M}$ such that either (1) or (2) below holds:

1. $\ell = \mathsf{A}\,\mathsf{B}!\mathsf{m}$ with $\mathfrak{q}(\mathsf{A}) \xrightarrow{\ell}_{\mathsf{A}} q'$ and
   a. $\mathfrak{q}' = \mathfrak{q}[\mathsf{A} \mapsto q']$
   b. $\mathfrak{b}' = \mathfrak{b}[\mathsf{A}\,\mathsf{B} \mapsto \mathfrak{b}(\mathsf{A}\,\mathsf{B}).\mathsf{m}]$
2. $\ell = \mathsf{A}\,\mathsf{B}?\mathsf{m}$ with $\mathfrak{q}(\mathsf{B}) \xrightarrow{\ell}_{\mathsf{B}} q'$ and
   a. $\mathfrak{q}' = \mathfrak{q}[\mathsf{B} \mapsto q']$ and
   b. $\mathfrak{b} = \mathfrak{b}'[\mathsf{A}\,\mathsf{B} \mapsto \mathsf{m}.\mathfrak{b}'(\mathsf{A}\,\mathsf{B})]$

in (1) m is sent on $\mathsf{A}\,\mathsf{B}$ while in (2) it is received. Machines and buffers not involved in the transition are left unchanged. We write $s \overset{\ell}{\Rightarrow} s'$ when s reaches s'.

Let S be a communicating system, a sequence $\pi = (s_i, \ell_i, s_{i+1})_{i \in I}$ where I is an initial segment of natural numbers (i.e., $i - 1 \in I$ for each $0 < i \in I$) is a run of S if $s_i \overset{\ell_i}{\Rightarrow} s_{i+1}$ is a transition of S for all $i \in I$. The set of runs of S is denoted as Δ_S^∞ and the set of runs of length k is denoted as Δ_S^k. Note that Δ_S^∞ may contain runs of infinite length, the set of finite runs of S is the union of all Δ_S^k and will be denoted as Δ_S. Given a run π, we define $\mathcal{L}[\pi]$ to be the sequence of labels $(\ell_i)_{i \in I}$. The *language* of S is the set $\mathcal{L}[S] = \{\mathcal{L}[\pi] \mid \pi \in \Delta_S^\infty\}$. Finally, *prf* $: \Delta_S^\infty \to 2^{\Delta_S}$ maps each run $\pi \in \Delta_S^\infty$ to its set of finite prefixes. As usual, for all $\pi \in \Delta_S^\infty$, the empty prefix ϵ belongs to *prf*(π).

The logic $\mathcal{QL}$ is introduced in [5] to express system-level QoS properties. Akin *DLTL* [15], $\mathcal{QL}$ is basically a linear temporal logic where atomic formulae, ranged over by ψ, constrain quantitative attributes, and the 'until' modality is restricted to specific runs. The syntax of $\mathcal{QL}$ is given by the grammar:

$$\Phi \ ::= \ \top \ \mid \ \psi \ \mid \ \neg\Phi \ \mid \ \Phi \vee \Phi \ \mid \ \Phi \ \mathsf{U}^{\mathsf{G}} \ \Phi$$

where $\top$ is the truth value 'true', $\neg$ and $\vee$ are the usual connectives for logical negation and disjunction, and the index G of the 'until' modality is a *global choreography* (g-choreographies for short) [6] meant to restrict the set of runs to be considered for the satisfiability of formulae[1]. G-choreographies can be thought of as regular expressions on the alphabet $\{\mathtt{break}\} \cup \{\mathsf{A} \to \mathsf{B} \colon \mathsf{m} \mid \mathsf{A}, \mathsf{B} \in \mathcal{P}, \mathsf{m} \in \mathcal{M}\}$, where $\mathtt{break}$ is used to stop iterations and $\mathsf{A} \to \mathsf{B} \colon \mathsf{m}$ represents

[1] Logical connectives $\wedge$ and $\implies$ are defined as usual while possibility $\langle \mathsf{G} \rangle \Phi$ and necessity $[\mathsf{G}]\Phi$ are defined as $\top \ \mathsf{U}^{\mathsf{G}} \ \Phi$ and $\neg\langle \mathsf{G} \rangle \neg\Phi$ respectively.

an interaction where A and B exchange message m. We let $\_ + \_$, $\_^*$, and $\_;\_$ respectively denote choice operator, Kleene star, and sequential composition (with $\_;\_$ taking precedence over $\_ + \_$).

Example 2. The g-choreography $\mathsf{G_{sys}} = \mathsf{A}\to\mathsf{B}\colon \mathsf{x}; \mathsf{B}\to\mathsf{A}\colon \mathsf{y}; \mathsf{G_{exch}}^*; \mathsf{A}\to\mathsf{B}\colon \mathsf{z2}$ corresponds to the qCS in Example 1 with $\mathsf{G_{exch}} = \mathsf{A}\to\mathsf{B}\colon \mathsf{z1}; \mathsf{B}\to\mathsf{A}\colon \mathsf{y}$ specifying the exchange of messages $\mathsf{z1}$ and y between A and B. $\diamond$

A g-choreography G induces a causality relation on the underlying communication whereby the output of an interaction precedes the corresponding input and, for the sequential composition $\mathsf{G}; \mathsf{G}'$ the actions in G precede those in G' when executed by a same participant. The *language* $\mathcal{L}[\mathsf{G}]$ of a g-choreography G consists of all possible sequences of communication actions compatible with the causal relation induced by G (note that $\mathcal{L}[\mathsf{G}]$ is prefix-closed). We write $\hat{\mathcal{L}}[\mathsf{G}]$ for the set of sequences in $\mathcal{L}[\mathsf{G}]$ that are not proper prefixes of any other sequence in $\mathcal{L}[\mathsf{G}]$. The definition of $\mathcal{L}[\mathsf{G}]$, immaterial here, can be found in [6]. The next example illustrates how to express a QoS property in $\mathcal{QL}$.

Example 3 (QoS properties). The runs of the system where where A and B exchange message $\mathsf{z1}$ and y three times can be specified by the g-choreography $\mathsf{G_3} = \mathsf{A}\to\mathsf{B}\colon \mathsf{x}; \mathsf{B}\to\mathsf{A}\colon \mathsf{y}; \mathsf{G_{exch}}; \mathsf{G_{exch}}; \mathsf{G_{exch}}$ where $\mathsf{G_{exch}}$ is defined in Example 2. Then the $\mathcal{QL}$ formula $\Phi \equiv [\mathsf{G_3}](\mathsf{cost} > 0) \implies [\mathsf{G_3}; \mathsf{G_{exch}}^*]\big(\mathsf{cost} \leqslant \mathsf{mem} \cdot 10\big)$ holds either if the first three exchanges do not have positive cost or if the cost of every subsequent exchange falls within the specified bounds. $\diamond$

The models of $\mathcal{QL}$ are defined in terms of runs of a QoS-extended communicating systems and an *aggregation function* [5] that formalises the conditions for a QoS property to hold in a run. The aggregation function, denoted below as agg_S depends on application-dependent binary *aggregation operators* that define how QoS attributes accumulate along a run. Hereafter, we assume that each QoS attribute has an associated aggregation operator.

A configuration is *accepting* if all participants are in an accepting state; a *completion* of a run π of a system S is a sequence π' ending in an accepting configuration such that $\pi\pi' \in \Delta_S^\infty$. A QoS property Φ is *satisfiable* in S if there exists a run $\pi \in \Delta_S^\infty$ with an accepting configuration such that $\langle \pi, \epsilon \rangle \models_S \Phi$ holds, where relation $\langle \_, \_ \rangle \models_S \_$ is defined as follows:

$$\begin{array}{rll}
\langle \pi, \pi' \rangle \models_S \psi & \text{iff} & \mathsf{agg}_S(\pi') \vdash_{RCF} \psi \quad \text{if } \psi \text{ atomic and } \pi' \in \mathit{prf}(\pi) \\
\langle \pi, \pi' \rangle \models_S \neg\Phi & \text{iff} & \langle \pi, \pi' \rangle \models_S \Phi \text{ does not hold} \\
\langle \pi, \pi' \rangle \models_S \Phi_1 \vee \Phi_2 & \text{iff} & \langle \pi, \pi' \rangle \models_S \Phi_1 \text{ or } \langle \pi, \pi' \rangle \models_S \Phi_2 \\
\langle \pi, \pi' \rangle \models_S \Phi_1 \ \mathsf{U}^{\mathsf{G}} \ \Phi_2 & \text{iff} & \text{there is a completion } \pi'' \text{ of } \pi' \text{ such that} \\
 & & \mathcal{L}[\pi''] \in \hat{\mathcal{L}}[\mathsf{G}],\ \langle \pi, \pi'\pi'' \rangle \models_S \Phi_2 \text{ and, for all} \\
 & & \pi''' \in \mathit{prf}(\pi''), \text{if } \pi''' \neq \pi'' \text{ then } \langle \pi, \pi'\pi''' \rangle \models_S \Phi_1.
\end{array}$$

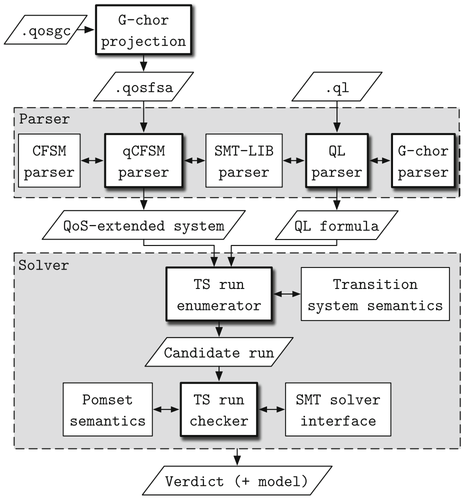
To handle atomic formulae, the first clause leverages *real-closed fields* (RCFs), a decidable formalisation of the first-order theory of real numbers [16, Thm. 37]. The 'until' modality requires Φ_2 to hold at some point along π, i.e. on a run $\pi'\pi''$ where completion π'' follow run π, with Φ_1 satisfied up to that point and

π'' compatible with G. A run π'' is compatible with a g-choreography G if it belongs to its language $\hat{\mathcal{L}}[\mathsf{G}]$.

A QoS property Φ is *valid* if, for all runs $\pi \in \Delta_S^\infty$ that contain an accepting configuration, $\langle \pi, \epsilon \rangle \models_S \Phi$. Given a QoS property Φ, a qCS S, and a bound k, the algorithm in [5] returns *true* when it finds a run $\pi \in \Delta_S^\infty$ of length at most k such that $\langle \pi, \epsilon \rangle \models_S \Phi$. Essentially, for each run of length at most k, the algorithm calls an auxiliary procedure that checks whether the run satisfies the QoS property by recursively following the definition of $\models_S$ presented above.

3 A Bounded Model Checker for QoS

We now present the architecture of MoCheQoS; a detailed presentation of its command line interface and the relevant file formats is in the accompanying artefact submission [11]. A graphical representation of the architecture is on the right, where tilted boxes represent files or data objects, rectangular boxes represent modules or functions, while arrows represent control and data flows. Thick shadowed boxes identify the modules developed in this work, while thin boxes identify ChorGram's modules and other open-source libraries used by MoCheQoS.



As ChorGram, MoCheQoS is implemented in Haskell. The two main modules are `Parser` and `Solver` (greyed dashed boxes). The former transforms the textual representations of MoCheQoS's inputs into internal Haskell representations used by the latter. More precisely, `qCFSM parser` leverages ChorGram's `CFSM parser` to process a textual description of the system from a `.qosfsa` file. Likewise, `QL parser` leverages `G-chor parser` (ChorGram's g-choreographies parser) to process a `.ql` file containing a simple textual description of the $\mathcal{QL}$ formula to verify. Both modules rely on the `SMT-LIB parser` (Haskell's `smt-lib` package).

The format of `.qosfsa` files is an extension of ChorGram's `.fsa` format. This extension enables the specification of the set of QoS attributes of interest with (*i*) aggregation operators, (*ii*) QoS constraints associated to the states of machines, and (*iii*) accepting states of machines. Additionally, MoCheQoS supports an extension of ChorGram's g-choreographies (a `.qosgc` file) to directly specify the QoS-related information over a g-choreography that can be projected on qCFSMs. This required to adapt ChorGram's `G-chor projection` and `G-chor parser` modules to support QoS specifications.

The `.ql` format borrows the g-choreography syntax of ChorGram for the case of the 'until' modality. The `Parser` module produces the `QoS-extended system` and the `QL formula` from the input files and passes them to `Solver`, our implementation of the bounded model checking algorithm in [5] (cf. Sect. 2). More precisely, `TS run enumerator` invokes `Transition system semantics` (the `TS` module of ChorGram) to enumerate the runs of the system that fall within the bound given in input parameter `k`. The enumeration is performed by systematically traversing the transition system. The process begins with the set of runs of length 1. Subsequently, the set of runs of length $i + 1$ is generated by appending all possible single-step transitions to each run of length i. Future work will explore heuristic-based approaches to traverse the transition system, aiming to prioritize the enumeration of potential counterexamples.

Then, `TS run enumerator` invokes `TS run checker` on each enumerated run to check if it is a model of the $\mathcal{QL}$ formula. Properties encompassing subformulas of the form $\Phi\ \mathsf{U}^{\mathsf{G}}\ \Phi'$ require `TS run checker` to invoke ChorGram's `PomsetSemantics` module in order to compute the language of G and to check membership of runs to it. To compute $\hat{\mathcal{L}}[\mathsf{G}]$, iterative subterms are replaced by their n-unfoldings, where $0 \leqslant n \leqslant \mathtt{u}$. The parameter `u` is configurable via the CLI of MoCheQoS and defaults to the value of parameter `k`. An n-unfolding of a subterm $\mathsf{G'}^{*}$ is defined as the sequential composition of G' with itself n times. The implementation of on-the-fly unfolding computation during transition system traversal is left for future work.

As recalled in Sect. 2, we leverage RCFs to express QoS constraints [5]; hence, we interfaced MoCheQoS with the state-of-the-art SMT solver Z3 [7] to check the validity of these constraints (i.e., $\vdash_{RCF}$). Properties involving QoS constraints of an atomic formulae require `TS run checker` to invoke the `SMT solver interface` to produce and check SMT-LIB queries. The `SMT solver interface` is composed of a modified version of Haskell's `smt-lib` package to build the SMT-LIB query and of Haskell's `SimpleSMT` package to call Z3. The SMT-LIB query produced for an atomic formula ψ allows to check whether there exists a counterexample to the entailment $\mathsf{agg}_S(\pi') \vdash_{RCF} \psi$. More precisely, the SMT-LIB query is structured as follows: (i) all quality attributes are declared as new symbols of sort `Real` using the `declare-const` command, and (ii) an `assert` statement is included with the expression $\mathsf{agg}_S(\pi') \wedge \neg\psi$. The construction of agg_S within the SMT-LIB query involves iterating through local states in run π'. For each state, the following steps are performed: (i) the corresponding QoS specification is collected, (ii) new symbols are declared for the local instantiation of quality attributes, and (iii) quality attribute symbols in the QoS specification are renamed to match the newly declared symbols. The SMT-LIB query is then sent to Z3 using Haskell's `SimpleSMT` package.

Finally, `Solver` returns a negative `Verdict` if the formula cannot be satisfied within the given bounds or a positive `Verdict` with a witnessing `model` (the run satisfying the `QL formula`) otherwise. To optimise computations, MoCheQoS maintains a balanced binary search tree to store the result of computing atomic entailments $\_ \vdash_{RCF} \_$, and a hash table to memorises the results of (a) the computation of the language of g-choreographies indexing 'until' operators, and (b) the membership check of a run to the language of a g-choreography.

4 Evaluation

Our empirical study aims to evaluate applicability and scalability of our approach. Towards applicability, Sect. 4.1 develops a case study adopting SLAs from the AWS cloud [12] while Sect. 4.2 borrows a case study from [13] to show how our approach can leverage automatic extraction of communicating systems. Towards scalability, Sect. 4.3 analyses MoCheQoS to measure its performance. Size and complexity of the case studies in Secs. 4.1 and 4.2 match what can be found in the literature (e.g., see [17–19]). As we will discuss later, the size of the models in Sect. 4.3 outmatches what can be found in system development.

The results presented in the next sections show that our framework can model SLAs present in industrial-strength scenarios (Sect. 4.1). Notably, MoCheQoS can effectively verify relevant system-level QoS in such scenarios and produce counterexamples useful to refine a property being checked. Moreover, Sect. 4.2 MoCheQoS can be used to effectively analyse system-level QoS properties of communicating systems automatically extracted from code.

4.1 SLA in the Amazon Cloud

The case study consists of a three-party version of the POP protocol [20] modelled after the OAuth authentication protocol [21]. More precisely, a client C securely accesses a remote mailbox server S after clearing authentication through a third party server A. This is specified by the g-choreography $\mathsf{G_{auth}} = \mathsf{C} {\rightarrow} \mathsf{A} : \mathsf{cred}\ ; (\mathsf{G_{token}} + \mathsf{A} {\rightarrow} \mathsf{C} : \mathsf{error})$ where $\mathsf{G_{token}}$ models the phase of the OAuth protocol where C acquires an authentication token granted if the credentials of the client C are valid; the acquired token allows C to prove its identity to the POP server S. This can be modelled as follows:

$$\begin{aligned}
\mathsf{G_{token}} &= \mathsf{A} {\rightarrow} \mathsf{C} : \mathsf{token} ; \mathsf{C} {\rightarrow} \mathsf{S} : \mathsf{token} ; (\mathsf{S} {\rightarrow} \mathsf{C} : \mathsf{fail} + \mathsf{S} {\rightarrow} \mathsf{C} : \mathsf{ok} ; \mathsf{G_{POP}}) \\
\mathsf{G_{POP}} &= \mathsf{G_{quit}} + \mathsf{C} {\rightarrow} \mathsf{S} : \mathsf{helo} ; \ \mathsf{S} {\rightarrow} \mathsf{C} : \mathsf{int} ; (\mathsf{G_{quit}} + \mathsf{G_{read}^{*}} \ ; \mathsf{G_{quit}}) \\
\mathsf{G_{quit}} &= \mathsf{C} {\rightarrow} \mathsf{S} : \mathsf{quit} ; \mathsf{S} {\rightarrow} \mathsf{C} : \mathsf{bye} \\
\mathsf{G_{read}} &= \mathsf{C} {\rightarrow} \mathsf{S} : \mathsf{read} ; \ \mathsf{S} {\rightarrow} \mathsf{C} : \mathsf{size} ; (\mathtt{break} + \mathsf{C} {\rightarrow} \mathsf{S} : \mathsf{retr} ; \ \mathsf{S} {\rightarrow} \mathsf{C} : \mathsf{msg} ; \ \mathsf{C} {\rightarrow} \mathsf{S} : \mathsf{ack})
\end{aligned}$$

We consider a system of qCFSMs, one per participant, realising the g-choreography $\mathsf{G_{auth}}$ (see [22, Appendix A] for the full model). The states of the qCFSMs are decorated by QoS specifications derived by publicly available SLAs. Specifically, we use the SLA of the Ory Network identity infrastructure [23] for A, the one for C reflects the SLA of clients of Amazon's Simple Email Service (SES) [24], and the SLA of S is modelled after the iRedMail service published in the AWS marketplace [25]. Our approach requires to constrain the quality attributes for each state of the participant while the constraints specified in the publicly available SLAs are relative to the whole execution. We overcome this obstacle by identifying the states in the qCFSMs which are relevant to the constraint. We then assign a corresponding QoS specification to each of these identified states. For example, the SLA of Amazon SES specifies that the price paid for each incoming email is 10^{-4} USD; this decorates the state in the client's qCFSM where mails are received.

Table 1. SLA attributes and parameters for the AWS case study

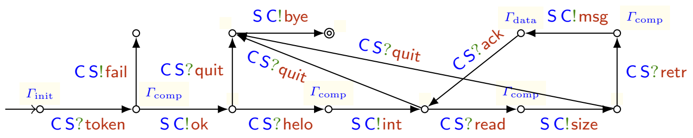
QoS attributes	pricing / configuration params.
num. of emails (emailsRetrieved)	price/hr for server software (hrRateSoftware)
data transferred out (Kb) (dataTransOut)	price/hr for infrastructure (hourlyRateCompute)
num. of authenticated users (usersAuth)	price/Gb for data transfer (transferGBRate)
price for incoming mails (priceEmails)	price/user for A (ratePerUserAuth)
server execution time (s) (execTimeServer)	num. of CPUs (CPUs)
total execution time (s) (execTime)	amount of memory (memCapacity)
	network performance (nwkPerf)
	instance type (instType)

We identified the quality attributes in Table 1 (left column) and the *pricing* and *configuration* parameters (right column in Table 1) that fix the elements of the computational infrastructure required. The value of the parameters are determined by the value of the instance type attribute, which models the type of the *compute* instance[2] chosen by the user when configuring the services. This relation is rendered in our model with logical implications. For example, AWS stipulates that if the selected instance type is 't4g.nano' (the smallest compute instance in the family 'T4g' represented as $\mathsf{instType} = 1$ in our model), the hourly rate for compute is 0.0042USD; this yields the implication

$$\mathsf{instType} = 1 \implies (\mathsf{hrRateCompute} = 0.0042 \wedge \mathsf{CPUs} = 2 \wedge \mathsf{memCapacity} = 0.5 \wedge \mathsf{nwkPerf} \leqslant 5)$$

to be included in the QoS specification of the initial state of the server qCFSM together with analogous implications for other instance types. The full model of our case study is an LTS with 34 configurations and 38 transitions obtained by the composition of three qCFSMs: the client C (15 states and 17 transitions), the server S (12 states and 14 transitions), and the authentication server A (4 states and 3 transitions). Note that CFSMs abstract away from local computations and focus only on the communication actions. Hence, the number of states and transitions only reflect the size of the communication protocol and not necessarily the size of the implementation. The QoS specifications we consider predicate over the 14 quality attributes in Table 1. Due to space limitations, here we only show the qCFSM for the server S (the other qCFSMs are in [22, Appendix A]):

$M_{\mathsf{S}} =$ (1)



where $\Gamma_{\text{comp}} = \{0.5 < \mathsf{execTime} < 3, \mathsf{execTimeServer} = \mathsf{execTime}\}$ and $\Gamma_{\text{data}} = \{10 < \mathsf{dataTransOut} < 500\}$, respectively modelling states where the server is performing significant computations and states where the server has sent data to

[2] In AWS jargon, *compute* refers to computational infrastructure, i.e., virtual computers that are rented through services like Amazon Elastic Compute Cloud (EC2).

the client. The specification Γ_{init} models the configuration of the AWS instance as stated earlier (see [22, Appendix A] for a full description). Let us focus on some system-level properties to be checked with MoCheQoS.

By inspecting the SLAs for AWS pricing scheme, we can derive the expression

$$\begin{aligned}\mathsf{totalCost} = \ & (\mathsf{execTime}/60^2)\cdot\mathsf{hrRateCompute} + (\mathsf{execTimeServer}/60^2)\cdot\mathsf{hrRateSoftware}\\ & + (\mathsf{dataTransOut}/1024^2)\cdot\mathsf{transferGBRate} + \mathsf{usersAuth}\cdot\mathsf{ratePerUserAuth} + \mathsf{priceEmails}\end{aligned}$$

for the overall cost of an execution of the system in terms of the aggregated values of the quality attributes, once the appropriate conversions are applied. We can then consider the $\mathcal{QL}$ formulae in Table 2 to check if the cost for receiving one email falls below a given threshold (Φ_1) and some relations between the total cost of an execution and the number of emails retrieved by the client (Φ_2, Φ_3, and Φ_4, which require to iterate the g-choreography $\mathsf{G}_{\mathsf{msg}}$). Both the validity of Φ_1 and a counterexample[3] for Φ_2 are computed by MoCheQoS in less than a second. In these cases it is not necessary to check for high values of the bound k because no iterative g-choreography occurs in Φ_1 and the counterexample of Φ_2 is found at $\mathsf{k} = 26$. The validity of the other formulae for a high value of k is checked by MoCheQoS in less than 3 min.

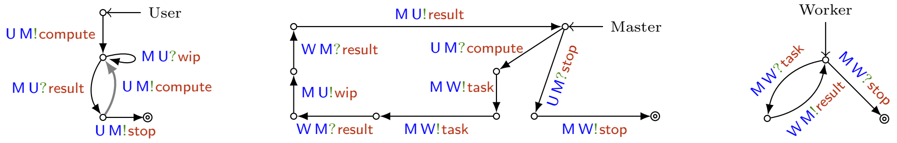
Table 2. Overall monetary cost of the coordinated execution of the three services

Let
$$\mathsf{G}_{\mathsf{init}} = \mathsf{C}\to\mathsf{A}: \mathsf{cred}; \mathsf{A}\to\mathsf{C}: \mathsf{token}; \mathsf{C}\to\mathsf{S}: \mathsf{token}; \mathsf{S}\to\mathsf{C}: \mathsf{ok}; \mathsf{C}\to\mathsf{S}: \mathsf{helo}; \mathsf{S}\to\mathsf{C}: \mathsf{int}$$
$$\mathsf{G}_{\mathsf{msg}} = \mathsf{C}\to\mathsf{S}: \mathsf{read}; \mathsf{S}\to\mathsf{C}: \mathsf{size}; \mathsf{C}\to\mathsf{S}: \mathsf{retr}; \mathsf{S}\to\mathsf{C}: \mathsf{msg}; \mathsf{S}\to\mathsf{C}: \mathsf{ack}$$

$\mathcal{QL}$-formula	k	Validity	MoCheQoS time
$\Phi_1 = [\mathsf{G}_{\mathsf{init}}; \mathsf{G}_{\mathsf{msg}}]\ \mathsf{totalCost} \leqslant 1$	26	✓	$< 1s$
$\Phi_2 = [\mathsf{G}_{\mathsf{init}}; \mathsf{G}_{\mathsf{msg}}{}^*]\ \mathsf{totalCost} \leqslant \mathsf{emailsRetrieved}$	26	CE	$< 1s$
$\Phi_3 = [\mathsf{G}_{\mathsf{init}}; \mathsf{G}_{\mathsf{msg}}{}^*]\ \mathsf{totalCost} \leqslant 1 + \mathsf{emailsRetrieved}$	100	✓	$135s$
$\Phi_4 = [\mathsf{G}_{\mathsf{init}}; \mathsf{G}_{\mathsf{msg}}{}^*]\ \mathsf{totalCost} \leqslant 0.5 + 0.5\cdot\mathsf{emailsRetrieved}$	100	✓	$135s$

4.2 Model Extraction

We show how to apply MoCheQoS on a model automatically inferred from the OCaml code of the case study in [13]. The system inferred in [13] is as follows[4]



The user requests the master to resolve a problem (whose nature is inconsequential here). The master splits the problem into two tasks and sends them to

[3] A run with 0 email retrievals, hinting at a fixed cost for executing the services.
[4] The thick gray arrow is the only addition we made to the original case study.

the worker, which replies to the master with the solutions of each task; between these replies, the master sends a 'work-in-progress' message to the user. Finally, the master sends the final result to the user by combining the partial results

The QoS specifications involve price, number of tasks computed, and allocated memory, respectively denoted with p, t, and mem. The contraints over these attributes have been manually specified and assigned to the states of the qCFSMs. For instance, we assume each problem instance (requested from the user) to require at most 5 units of memory and model this by adding a contraint over mem to the states where memory is allocated for the problem intance. Similarly, we assume the result of the problem to require at most 1 unit of memory. Additionally, we assume that the master charges a flat fee of 10 monetary units once the computation is completed, while the worker's cost varies based on the size of the task. We check the following $\mathcal{QL}$ properties:

$\Phi_1 \equiv [\mathsf{G}](\mathsf{t} \cdot 6 \leqslant \mathsf{p} < 12.5)$ $\quad\Phi_3 \equiv [\mathsf{G}^*](1 \leqslant \mathsf{mem} < 10)$

$\Phi_2 \equiv [\mathsf{G}^*](\mathsf{t} \cdot 6 \leqslant \mathsf{p} < 12.5)$ $\quad\Phi_4 \equiv (\mathsf{p} \leqslant \mathsf{t} \cdot 12.5)\ \mathsf{U}^{\mathsf{G}}\ ([\mathsf{G}^*]\ \mathsf{p} \leqslant 25)$

where G describes the process of computing one problem instance, starting with $\mathsf{U}{\rightarrow}\mathsf{M}$: compute and ending with $\mathsf{M}{\rightarrow}\mathsf{U}$: result ([22, Appendix C] reports the details about the models of this case study.) Formula Φ_1 uses the necessity modality to express bounds on the price of the computation of one problem instance. Formulas Φ_2 and Φ_3 use the necessity modality to express bounds on the price and the memory used after computing any number of problem instances. Formula Φ_4 states that (i) up to the computation of the first problem instance, the price falls below a bound depending on the number of tasks computed, and (ii) afterwards, the price is always bounded by 25 right after any number of computed problem instances. We applied MoCheQoS on these formulas with bounds that correspond to unfolding loops once and twice ($\mathsf{k} = 18$ and $\mathsf{k} = 32$ are, respectively, the lengths of runs where the master sends the result of one and two problem instances and the user stops). The results of the experiments are summarised in Table 3 where times are in seconds. Noticeably, for satisfiability the results with $\mathsf{k} = 18$ subsume those with $\mathsf{k} = 32$; also, for Φ_4, a bound of 32 is needed to find a counterexample, which shows that the formula is satisfiable but not valid.

Table 3. Results on model extraction case study (CE = counter example)

Formula	Bound k = 18				Bound k = 32			
	satisfiability		validity		satisfiability		validity	
	Time (s)	Result	Time (s)	Result	Time (s)	Result	Time (s)	Result
Φ_1	.3	sat	1.7	No CE	.3	sat	34	No CE
Φ_2	.3	sat	1.9	No CE	.3	sat	185	No CE
Φ_3	.3	sat	1.9	No CE	.3	sat	186	No CE
Φ_4	.5	sat	6	No CE	.6	sat	30	CE

4.3 Performance

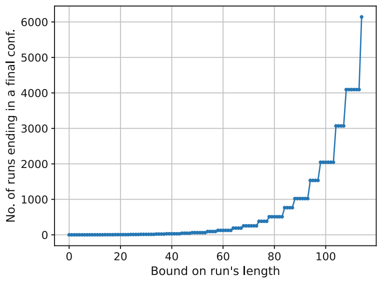
The performance of MoCheQoS depends on the cost of checking if a formula Φ holds on any run of the system S of length at most `k`; this cost is dominated by the evaluation of the 'until' sub-formulae[5] $\Phi_1 \ \mathsf{U}^{\mathsf{G}} \ \Phi_2$ which depends on the complexity of G and of Φ_1 and Φ_2. We therefore generate synthetic properties following the pattern $\Phi \equiv \Phi_1 \ \mathsf{U}^{\mathsf{G}} \ \Phi_2$ and varying the size and complexity of G. Formulas Φ_1 and Φ_2 are created to cover (i) the best case (any run that matches the language of G satisfies Φ), (ii) the worst case (no run matching the language of G satisfies Φ), and (iii) the 'average' case (only a single random run that matches the language of G satisfies Φ).

The performance analysis of MoCheQoS was driven by experiments[6] tailored to address the following questions:

Loop unfolding. How does performance evolve as we increase the number of loop unfoldings in g-choreographies indexing 'until' sub-formulae?

Nested choices. How does average performance evolve as we increase the number of nested choices in g-choreographies indexing 'until' sub-formulae?

Loop unfolding. The experiments are performed on the set of qCFSMs described in the AWS cloud case study in Sect. 4.1. To help the reader understand the size of the problem, we show below the number of runs of this system as a function of the bound `k` on the size of the runs. Due to interleaving of the transitions in the asynchronous communication, the number of runs grows exponentially when the number of loop unfoldings in the POP protocol increases.

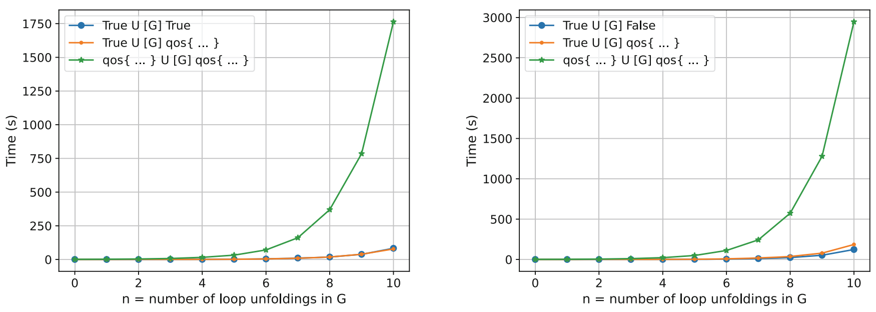


We synthetically generate six families of $\mathcal{QL}$ formulas with the shape $\Phi_1 \ \mathsf{U}^{\mathsf{G}} \ \Phi_2$ where $\mathsf{G} = \mathsf{G}_{\mathsf{init}}; {\mathsf{G}_{\mathsf{msg}}}^{n}$, for $1 \leqslant n \leqslant 10$, with $\mathsf{G}_{\mathsf{init}}$ and $\mathsf{G}_{\mathsf{msg}}$ defined as in Sect. 4.1. The first three families of formulas are constructed to be satisfiable while the last three to be unsatisfiable. The unsatisfiable formulas are constructed by guaranteeing that no run compatible with G satisfies Φ_2. For each formula we execute MoCheQoS with a bound $\mathtt{k} = 16 + 10n$, which guarantees that the runs of the system that match G are reached. The results are shown in Fig. 1. Figure 1a plots the time it takes for MoCheQoS to find a model for the three families of satisfiable formulas as a function of n. The three families differ in how Φ_1 and Φ_2 are constructed: (i) both as atomic truth values, (ii) Φ_1 as an atomic truth value and Φ_2 as a QoS constraint, or (iii) both as QoS constraints. Figure 1b plots the time MoCheQoS takes to report that no model was found for the three families of unsatisfiable

[5] This requires to query the SMT-LIB to solve QoS constraints; we use Z3 as a blackbox which we cannot control; therfore, its computational costs are factored out.

[6] We used Z3 v4.10.2 and an 8-cores MacBook Pro (Apple M1) with 16 GB of memory.

formulas as a function of n. The results show that the main source of computational burden, as the number of loop unfoldings increases, is the verification of the QoS constraint in Φ_1, the first operand of the 'until' operator. In Fig. 1a and Fig. 1b, this is manifested by the green line growing significantly faster than the other two lines. The explanation for this is that, due to the semantics of the 'until', the verification of Φ_1 has to be performed in every prefix of the run and, when Φ_1 is a QoS constraint, each verification is done by calling Z3 with a different SMT-LIB query.



(a) Satisfiable instances (b) Unsatisfiable instances

Fig. 1. Execution time to analyse (un)satisfiable 'until' formulas

Nested choices. To evaluate the performance of MoCheQoS in the presence of nested choices indexing 'until' sub-formulae, we will construct synthetic systems by varying the number of nested choices in the g-choreography of the system. We consider systems of two participants taking turns in sending a message to each other; the sender of each turn chooses between two messages. Due to the branching nature of this behaviour, the number of runs in a system grows as 2^n where n is the number of nested choices (i.e., the number of turns). We synthetically generate systems with this behaviour by varying n from 1 to 10. Remarkably, nested choices correspond to nested conditional statements and accepted metrics recommend to keep low the nesting level of conditional statements. In particular, an accepted upper bound of *cyclomatic complexity*[7] is 15, which corresponds to less than 4 nested conditional statements.

To generate these systems, we craft QoS-extended g-choreography in `.qosgc` format and then leverage ChorGram's `G-chor projection` to obtain the qCFSMs of the system. The QoS specifications comprise five QoS attributes and determine unique values for them in each accepting state, enabling the construction of $\mathcal{QL}$ formulas that are satisfied by only one run of the system. In this way, we can use these generated cases to evaluate the performance of MoCheQoS in finding the only run that satisfies a formula in a search space of exponential size in n.

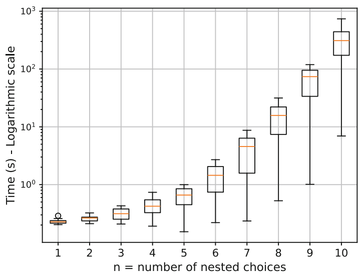
[7] Cyclomatic complexity [26] measures the complexity of programs according to the number of independent paths represented in the source code.

```
Bob → Alice:  m0  ; {
    Alice → Bob:  m0;  Bob → Alice:  leaf1
    +
    Alice → Bob:  m1;  Bob → Alice:  leaf2
}

+

Bob → Alice:  m1;  {
    Alice → Bob:  m0;  Bob → Alice:  leaf3
    +
    Alice → Bob:  m1;  Bob → Alice:  leaf4
}
```

(a) Generated g-choreography for $n = 2$



(b) Average execution time versus n

Fig. 2. Performance on the ‘average’ case for formulas with nested choices

The formula is generated by following the pattern $\top \;\; \mathsf{U}^{\mathsf{G}} \;\; \psi$, were G matches every run of the system and ψ is a QoS constraint, determining the value of the five QoS attributes, that is satisfied by only one run. Figure 2a shows the generated G for two nested choices. See [22, Appendix B] for a detailed view of the files used in this case study. The bound k is set high enough to guarantee that all runs of the system are reached by the analysis. For each value of n, we generate 100 different random instances of the $\mathcal{QL}$ formula, where the only run that satisfies the formula is chosen randomly, and execute MoCheQoS on each instance. Figure 2b shows the results as a boxplot per number of nested choices. Remarkably, both the average execution time and its variance grow as n increases. This is due to the fact that the difference in time between the best and worst case scenarios, where the model is found either in the first or the last enumerated run matching G, increases with n. The apparent bias in dispersion towards lower execution times is just a visual effect due to the logarithmic scale of the y-axis.

5 Related Work

We position MoCheQoS in the category of static analysers of system-level QoS properties. There is a vast literature on QoS, spanning a wide range of contexts and methods [27,28], QoS for choreographies [29,30], and formal models and analysis procedures that have been proposed without tool supported analysis.

A tool for the automatic analysis of QoS properties appeared in [31] where QoS specifications were expressed as theory presentations over quantitative attributes but only considering convex polytopes; this restriction is not present in our language. Unlike MoCheQoS, the approach in [31] relies on “monolithic” specifications of QoS, rendering hard its application to distributed systems without adding some composition mechanisms. We instead assign QoS contracts to states of communicating services and then aggregate them in order to analyse properties along executions of the whole system.

The Envisage project [32] aims to provide a framework for the development and deployment of virtualized scalable services in the cloud. System design and analysis is done in ABS [33,34] which enables resource-awareness in design and deployment decisions [35–37]. In ABS resource awareness revolves around the concept of elasticity and the capability of simulating the execution of the system under deployment constraints. The analysis is performed to evaluate bounds over fixed attributes (*Speed*, *Bandwidth*, *Memory*, and *Cores*) in order to inspect consumption of those resources over simulated executions. On the contrary, MoCheQoS is agnostic to the attribute's nature, as far as it is measurable, and provides a static analysis procedure capable of exploring the whole execution space (with respect to a predefined bound to the trace length) in the search for counterexamples of arbitrary dynamic temporal properties, not only bounds.

Metric functions are used in [38] to verify SLAs of client-server systems via the interactive theorem prover KeY [39]. We can deal with multiparty system and the analysis of QoS properties of MoCheQoS is fully automatic. Other abstract models of QoS such as quantales [40] or c-semirings [41–43] have been proposed. Process calculi capable of expressing SLAs appeared in [41] and in [43] without a specific analysis technique. A variant of the μ-calculus equipped with the capability of expressing QoS properties and an analysis algorithm has been presented in [42] without an implementation.

Automatic extraction of local QoS contracts from global QoS specifications is defined and implemented in [29]; the paper proposes applications including the use of the derived contracts for monitoring but no static analysis procedure of the QoS systems' behaviour is proposed. On the same basis, monitoring algorithms were presented in [30] and contracts are used for run-time prediction, adaptive composition, or compliance checking.

Probabilistic model checking (PMC) [44,45] implemented in PRISM [46] features the automated analysis of quantitative properties. The main differences with respect to our work are the modelling language and the properties that can be checked. First, PMC models are usually expressed as Markov chains while MoCheQoS does not feature probabilistic information. Second, RCFs are more expressive than the reward functions adopted in [44] since they allow to express first order formulae over QoS attributes. For example, the QoS specifications shown in Sect. 4.1 cannot be expressed with PRISM's reward functions. Finally, while in PMC properties are expressed as temporal formulae over bounds on the expected cumulative value of a reward, MoCheQoS can verify dynamic temporal formulae where atoms are first order formulae over QoS attributes and temporal modalities are indexed with g-choreographies. Our setting leads to the computation of an aggregation function that collects QoS specifications of states along a run, which is not the case in PMC. Timed automata [47] are used in UPPAAL [48] to verify real-time systems. Our QoS specifications can predicate about time but, unlike in UPPAAL, the behaviour of systems is independent of it. It is therefore not straightforward to compare MoCheQoS with tools like PMC or UPPAAL as they are designed for different purposes. Extending MoCheQoS with time and probabilities is indeed an intriguing endeavour.

6 Conclusions and Future Work

We presented MoCheQoS, a tool to verify QoS properties of message-passing systems. We build a bounded model checker upon the dynamic logic and semi-decision procedure recently presented in [5] which rely on choreographic models. To our best knowledge, MoCheQoS is the first tool to support the static analysis of QoS for choreographic models of message-passing systems. The satisfiability of QoS constraints in atomic formulas is delegated to the SMT solver Z3 while ChorGram is used to handle the choreographic models and their semantics. Notably, MoCheQoS can handle any quality attribute that takes values in the real numbers (if it is equipped with an appropriate aggregation operator), making it highly versatile. Experiments to evaluate the applicability of our approach were conducted over case studies based on the AWS cloud and on models automatically extracted from code. Experiments to evaluate the scalability of our approach were conducted over synthetically generated models and properties.

Our experiments demonstrate the effectiveness of MoCheQoS. Nevertheless, there is room for improvement. We are considering abstract semantics where runs are partitioned in equivalence classes so that we have to check only representative runs of such classes in order to tackle the computational blow up due to asynchronous communications as discussed in Sect. 4.

In scope of future work is also the definition of a domain-specific language to ease the modelling phase. For instance, such language could feature data types to express non-cumulative attributes (as those used in Sect. 4.1).

Data Availability. The source code of the tool, the code for generating the experimental data, and detailed instructions on how to reproduce the results presented in this paper are available in the Zenodo repository with the identifier https://doi.org/10.5281/zenodo.10038447. (See citation in [11]).

References

1. Fiadeiro, J.L., Lopes, A., Bocchi, L.: An abstract model of service discovery and binding. Formal Aspects Comput. **23**(4), 433–463 (2011)
2. Amazon: AWS Lambda Service Level Agreement. https://aws.amazon.com/lambda/sla/
3. Amazon: AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/
4. World Wide Web Consortium: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. https://www.w3.org/TR/wsdl20/
5. Lopez Pombo, C.G., Martinez Suñé, A.E., Tuosto, E.: A dynamic temporal logic for quality of service in choreographic models. In: Ábrahám, E., Dubslaff, C., Tarifa, S.L.T., eds.: Proceedings of 20th International Colloquium on Theoretical Aspects of Computing - ICTAC 2023. Volume 14446 of Lecture Notes in Computer Science., Lima, Perú, Springer-Verlag (December 2023), pp. 119–138 (2023). https://doi.org/10.1007/978-3-031-47963-2_9
6. Tuosto, E., Guanciale, R.: Semantics of global view of choreographies. J. Logical Algebraic Methods. Program. **95**, 17–40 (2018)

7. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C., Rehof, J., eds.: Proceedings of 14th International Conference Tools and Algorithms for the Construction and Analysis of Systems TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008. Volume 4963 of Lecture Notes in Computer Science., Springer-Verlag (2008), pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Coto, A., Guanciale, R., Tuosto, E.: Choreographic development of message-passing applications - A tutorial. In Bliudze, S., Bocchi, L., eds.: Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings. Volume 12134 of Lecture Notes in Computer Science., Springer (2020) 20–36
9. Coto, A., Guanciale, R., Lange, J., Tuosto, E.: ChorGram: tool support for choreographic development (2015). https://bitbucket.org/eMgssi/chorgram/src/master/
10. Lange, J., Tuosto, E., Yoshida, N.: A tool for choreography-based analysis of message-passing software. In: Gay, S., Ravara, A., eds.: Behavioural Types: from Theory to Tools. Automation, Control and Robotics. River (2017), pp. 125–146 (2017)
11. Lopez Pombo, C.G., Martinez-Suñé, A.E., Tuosto, E.: MoCheQoS: Automated Static Analysis of Quality of Service Properties of Communicating Systems - Artifact. https://zenodo.org/doi/10.5281/zenodo.10038447. Git repository available at https://bitbucket.org/aemartinez/chorgram/src/mocheqos-fm24/. June 2024
12. Amazon: AWS Global Infrastructure. https://aws.amazon.com/about-aws/global-infrastructure. Accessed 27 March 2024
13. Imai, K., Lange, J., Neykova, R.: Kmclib: automated inference and verification of session types from OCaml programs. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 379–386. Cham, Springer International Publishing, Lecture Notes in Computer Science (2022)
14. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983)
15. Henriksen, J.G., Thiagarajan, P.: Dynamic linear time temporal logic. Ann. Pure Appl. Logic **96**(1–3), 187–207 (1999). Originally published in [49]
16. Tarski, A.: A decision method for elementary algebra and geometry. Memorandum RM-109, RAND Corporation (1951) Later published in [50] and reprinted in [51]
17. Albert, E., et al.: Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. Serv. Oriented Comput. Appl. **8**(4), 323–339 (2014). This article refines and substantially extends [52]
18. Iraci, G., Chuang, C.E., Hu, R., Ziarek, L.: Validating IoT devices with rate-based session types. Proc. ACM Program. Lang. **7**(OOPSLA2), 1589–1617 (2023)
19. Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. In: Dillig, I., Tasiran, S., eds.: Proceedings of 31st International Conference Computer Aided Verification (CAV 2019), Part I. Volume 11561 of Lecture Notes in Computer Science., Springer-Verlag (July 2019), pp. 97–117 (2019). https://doi.org/10.1007/978-3-030-25540-4_6
20. Anonymous: Post office protocol: Version 2 (1985). https://rfc-editor.org/rfc/rfc937.txt
21. Hardt, D.: The OAuth 2.0 Authorization Framework (2012). https://rfc-editor.org/rfc/rfc6749.txt

22. Lopez Pombo, C.G., Martinez Suñé, A.E., Tuosto, E.: MoCheQoS: automated analysis of quality of service properties of communicating systems. On-line (November 2023) https://arxiv.org/abs/2311.01415
23. Ory: Ory - API-first Identity Management, Authentication and Authorization. For Secure, Global, GDPR-compliant Apps. https://www.ory.sh/. Accessed 3 April 2024
24. Amazon Web Services, Inc.: Bulk Cloud Email Service - Amazon Simple Email Service - AWS. https://aws.amazon.com/ses/. Accessed 3 April 2024
25. Amazon Web Services Inc.: AWS Marketplace: iRedMail (IMAP, SMTP, POP3) Email Server on Ubuntu 18.04 LTS. https://aws.amazon.com/marketplace/pp/prodview-xswbskbnidz5e. Accessed 3 April 2024
26. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **SE-2**(4), 308–320 (1976)
27. Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., Meedeniya, I.: Software architecture optimization methods: a systematic literature review. IEEE Trans. Software Eng. **39**, 658–683 (2013)
28. Hayyolalam, V., Kazem, A.A.P.: A systematic literature review on QoS-aware service composition and selection in cloud environment. J. Netw. Comput. Appl. **110**, 52–74 (2018)
29. Ivanović, D., Carro, M., Hermenegildo, M.V.: A constraint-based approach to quality assurance in service choreographies. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q., eds.: Proceedings of 10th International Conference on Service-Oriented Computing – ICSOC 2012. Volume 7636 of Lecture Notes in Computer Science., pp. 252–267. Springer-Verlag (November 2012). https://doi.org/10.1007/978-3-642-34321-6_17
30. Kattepur, A., Georgantas, N., Issarny, V.: Qos analysis in heterogeneous choreography interactions. In: Basu, S., Pautasso, C., Zhang, L., Fu, X., eds.: Proceedings of the 11nd International Conference on Service Oriented Computing – ICSOC '13. Volume 8274 of Lecture Notes in Computer Science., Springer-Verlag (December 2013), pp.23–38 (2013). https://doi.org/10.1007/978-3-642-45005-1_3
31. Martinez Suñé, A.E., Lopez Pombo, C.G.: Automatic quality-of-service evaluation in service-oriented computing. In: Nielson, H.R., Tuosto, E., eds.: Proceedings of Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019. Volume 11533 of Lecture Notes in Computer Science., Springer-Verlag (June 2019), pp. 221–236 (2019). https://doi.org/10.1007/978-3-030-22397-7_13
32. Envisage: Engineering Virtualized Services. http://envisage-project.eu
33. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M., eds.: Revised papers of the 9th International Symposium Formal Methods for Components and Objects (FMCO 2010). Volume 6957 of Lecture Notes in Computer Science., Springer-Verlag (2012), pp. 142–164 (2012)
34. The ABS Framework. https://abs-models.org, https://github.com/abstools/abstools
35. de Gouw, S., Mauro, J., Nobakht, B., Zavattaro, G.: Modeling deployment decisions for elastic services with ABS. On-line (2016). http://www.envisage-project.eu/modeling-deployment-decisions-for-elastic-services-with-abs/
36. de Gouw, S., Mauro, J., Nobakht, B., Zavattaro, G.: Declarative elasticity in ABS. In: Aiello, M., Johnsen, E.B., Dustdar, S., Georgievski, I., eds.: Proceedings of

Service-Oriented and Cloud Computing (ESOCC 2016) - 5th IFIP WG 2.14 European Conference. Volume 9846 of Lecture Notes in Computer Science., Springer-Verlag (September 2016), pp. 118–134 (2016). https://doi.org/10.1007/978-3-319-44482-6_8
37. de Boer, F.S., et al.: Analysis of SLA compliance in the cloud - an automated, model-based approach. In: Ancona, D., Pace, G., eds.: Proceedings of Second Workshop on Verification of Objects at RunTime EXecution (VORTEXECOOP/ISSTA 2018). Volume 302 of EPTCS. (July 2019) pp. 1–15 (2019)
38. Giachino, E., de Gouw, S., Laneve, C., Nobakht, B.: Statically and dynamically verifiable SLA metrics. In: Ábrahám, E., Bonsangue, M.M., Johnsen, E.B., eds.: Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday. Volume 9660 of Lecture Notes in Computer Science., Springer (2016), pp. 211–225 (2016). https://doi.org/10.1007/978-3-319-30734-3_15
39. Din, C.C., Bubel, R., Hähnle, R.: Key-abs: a deductive verification tool for the concurrent modeling language ABS. In Felty, A.P., Middeldorp, A., eds.: Proceedings of 25th International Conference on Automated Deduction - CADE-25. Volume 9195 of Lecture Notes in Computer Science., Springer-Verlag (August 2015), pp. 517–526 (2015). https://doi.org/10.1007/978-3-319-21401-6_35
40. Rosenthal, K.I.: Quantales and their application. Volume 234 of Pitman research notes in mathematics series. Longman Scientific & Technical (1990)
41. Buscemi, M.G., Montanari, U.: Cc-pi: a constraint-based language for specifying service level agreements. In: De Nicola, R., ed.: Proceedings of 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2007). Volume 4421 of Lecture Notes in Computer Science., Springer-Verlag (2007), pp. 18–32 (2007). https://doi.org/10.1007/978-3-540-71316-6_3
42. Lluch-Lafuente, A., Montanari, U.: Quantitative μ-calculus and CTL based on constraint semirings. Electron. Notes Theoret. Comput. Sci. **112**, 37–59 (2005). Proceedings of the Second Workshop on Quantitative Aspects of Programming Languages (QAPL 2004)
43. De Nicola, R., Ferrari, G., Montanari, U., Pugliese, R., Tuosto, E.: A process calculus for QoS-aware applications. In: Jacquet, J.M., Picco, G.P., eds.: Proceedings of 7th International Conference Coordination Models and Languages, COORDINATION 2005. Volume 3454 of Lecture Notes in Computer Science., Springer-Verlag (April 2005), pp. 33–48 (2005). https://doi.org/10.1007/11417019_3
44. Kwiatkowska, M.: Quantitative verification: models, techniques and tools. In: The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers. ESEC-FSE Companion '07, New York, NY, USA, Association for Computing Machinery (September 2007), pp. 449–458 (2007)
45. Baier, C., Katoen, J.P., eds.: Principles of Model Checking. The MIT Press (2008)
46. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification, pp. 585–591. Lecture Notes in Computer Science, Berlin, Heidelberg, Springer (2011)
47. Alur, R., Dill, D.L.: A theory of timed automata. Theoret. Comput. Sci. **126**(2), 183–235 (1994)
48. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Int. J. Softw. Tools Technol. Transfer **1**, 134–152 (1997)

49. Henriksen, J.G., Thiagarajan, P.: Dynamic linear time temporal logic. Report Series BRICS-RS-97-8, Basic Research in Computer cience (1997). https://tidsskrift.dk/brics/issue/view/2365
50. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. University of California Press (1951) Originally published in [16] and reprinted in [51]
51. Tarski, A.: A Decision method for elementary algebra and geometry. In: Caviness, B.F., Johnson, J.R. (eds) Quantifier Elimination and Cylindrical Algebraic Decomposition. Texts and Monographs in Symbolic Computation. Springer, Vienna (1998). Originally published in [16] and reprinted from [50]. https://doi.org/10.1007/978-3-7091-9459-1_3
52. de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Wong, P.Y.: Formal modeling of resource management for cloud architectures: an industrial case study. In: Paoli, F.D., Pimentel, E., Zavattaro, G., eds.: Proceedings of First European Conference Service-Oriented and Cloud Computing (ESOCC 2012). Volume 7592 of Lecture Notes in Computer Science., Springer-Verlag (September 2012), pp. 91–106 Refined and substantially extended in [17]. https://doi.org/10.1007/978-3-642-33427-6_7

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Alloy Repair Hint Generation Based on Historical Data

Ana Barros[1,3], Henrique Neto[2,3], Alcino Cunha[2,3], Nuno Macedo[1,3](✉), and Ana C. R. Paiva[1,3]

[1] Universidade do Porto, Porto, Portugal
{nmacedo,apaiva}@fe.up.pt
[2] Universidade do Minho, Braga, Portugal
alcino@di.uminho.pt
[3] INESC TEC, Porto, Portugal

Abstract. Platforms to support novices learning to program are often accompanied by automated next-step hints that guide them towards correct solutions. Many of those approaches are data-driven, building on historical data to generate higher quality hints. Formal specifications are increasingly relevant in software engineering activities, but very little support exists to help novices while learning. Alloy is a formal specification language often used in courses on formal software development methods, and a platform—Alloy4Fun—has been proposed to support autonomous learning. While non-data-driven specification repair techniques have been proposed for Alloy that could be leveraged to generate next-step hints, no data-driven hint generation approach has been proposed so far. This paper presents the first data-driven hint generation technique for Alloy and its implementation as an extension to Alloy4Fun, being based on the data collected by that platform. This historical data is processed into graphs that capture past students' progress while solving specification challenges. Hint generation can be customized with policies that take into consideration diverse factors, such as the popularity of paths in those graphs successfully traversed by previous students. Our evaluation shows that the performance of this new technique is competitive with non-data-driven repair techniques. To assess the quality of the hints, and help select the most appropriate hint generation policy, we conducted a survey with experienced Alloy instructors.

Keywords: formal specification · intelligent tutoring system · automated hints · Alloy

1 Introduction

Formal specification languages are based on mathematical formalisms and are used to describe the expected behaviour of a software component. Formal specifications are increasingly embraced by software engineering professionals, in *lightweight* formal development techniques such as automated synthesis, testing or monitoring. Moreover, they will possibly become even more relevant as

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 104–121, 2025.
https://doi.org/10.1007/978-3-031-71177-0_8

advances in large language models push programming activities into higher levels of abstraction [29].

Alloy [12,13] is a formal specification language that allows the automatic analysis of software design models with rich structure and behaviour. Due to its high-level of abstraction, flexibility and simplicity, Alloy is often used in introductory formal methods courses[1]. Yet, studies show that novices, and even experienced professionals, struggle with understanding and writing Alloy specifications [17]. The Alloy4Fun [16] web platform was developed in this educational context to ease the sharing of specification challenges with auto-grading, supporting instructors in classes and allowing students to study autonomously. Intelligent tutoring systems (ITS) for programming have long relied on automated feedback to support students in large classes and outside the classroom. Alloy4Fun, like regular Alloy, is solver-based and provides feedback for incorrect specifications as graphical counter-examples. This is a popular feature among Alloy practitioners and could, in principle, act as hints to help students progress towards solving a challenge when learning autonomously. However, studies find visual counter-examples have mixed results with novices [7,8]. In fact, a recent user study [6] with different kinds of manually encoded hints concluded that only *next-step hints*, which highlight faults in incorrect specifications and provide tips on how to fix them, improved the immediate performance of the participants without jeopardizing learning retention.

Next-step hints are one of the most common feedback approaches in ITSs for programming [21]. A possible approach to generate these hints is through automated repair techniques. After repairing a faulty program to obtain a correct one, a next-step hint can be obtained by comparing both. One such technique has been proposed for Alloy [4], but it is only effective when students are already close to a correct specification, and the quality of the generated hints is not clear. An alternative approach is to rely on historical student submission data for the generation of hints, in order to guide the student towards paths that led to successful submissions. The expectation is that more understandable hints can be generated by mimicking successful peer behaviour.

This work proposes the first history-based hint-generation technique for Alloy, and presents its implementation as an extension to Alloy4Fun. Alloy4Fun was also designed to support research on formal methods education, and thus every interaction with the tool is anonymously recorded and made available to the instructors [16]. Based on this collected data, the proposed extension creates a directed graph encoding all attempts by previous students. Then, upon a hint request, it finds a path between the student submission and a solution using a customizable policy, and generates a next-step hint based on this path. The developers of Alloy4Fun maintain a publicly available dataset [15] of student attempts collected from their classes over the years. We relied on this dataset to evaluate our technique both for performance (effectiveness and efficiency) and for the quality of the hints (based on the opinions of experts on teaching Alloy). It achieved better results than the state-of-the-art tools. Furthermore, it can generate timely

[1] http://alloytools.org/citations/courses.html.

```
sig User {
   follows : set User,
   posts   : set Photo
}
sig Influencer extends User {}
sig Photo {
   date    : one Day
}
sig Ad extends Photo {}
sig Day {}

/* Every photo is posted by one user. */
pred spec1 {
}
//SECRET
pred oracle1 { all p: Photo | one posts.p }
//SECRET
check spec1 { oracle1 iff spec1 } for 3

/* Influencers are followed by everyone else. */
pred spec2 {
}
//SECRET
pred oracle2 { all u:User | Influencer - u in u.follows }
//SECRET
check spec2 { oracle2 iff spec2 } for 3
```

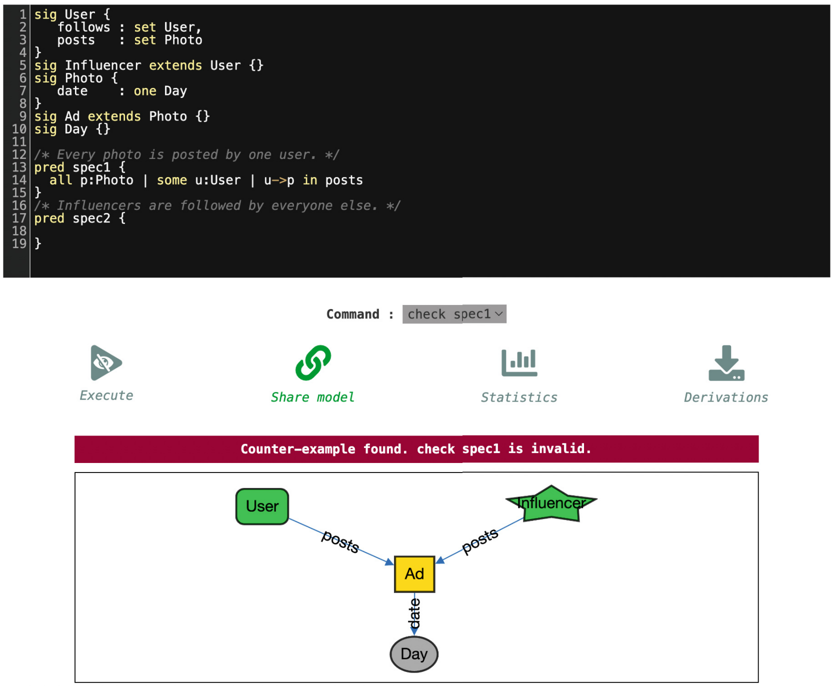
Fig. 1. Social network model with specification challenges

feedback, which is especially important in the educational context since students might easily feel frustrated if hints take too long to generate.

The remainder of the paper is structured as follows. Section 2 provides a short introduction to Alloy education, and Sect. 3 describes techniques for hint generation and Alloy repair. Section 4 presents our solution and its implementation, which is evaluated in Sect. 5. Section 6 presents conclusions and future work.

2 Teaching Alloy with Alloy4Fun

The Alloy language is based on temporal relational logic, but for simplicity, we'll restrict this presentation to the static subset of the language. Structure in an Alloy model is introduced through the declaration of *signatures* and *fields*. These can be restricted by multiplicity constraints and be hierarchically organized. The upper part of Fig. 1 depicts the structure of a social network system, a simplified version of an exercise in the Alloy4Fun dataset [15]. A signature `User` models users, with binary fields `follows` and `posts` relating each user with a **`set`** of users being followed, and a **`set`** of posted photos, respectively. Signature `Influencer` extends users, denoting a subset of `User`. Signature `Photo` has a field `date` that



```
sig User {
   follows : set User,
   posts   : set Photo
}
sig Influencer extends User {}
sig Photo {
   date    : one Day
}
sig Ad extends Photo {}
sig Day {}

/* Every photo is posted by one user. */
pred spec1 {
  all p:Photo | some u:User | u->p in posts
}
/* Influencers are followed by everyone else. */
pred spec2 {

}
```

Command : check spec1

Counter-example found. check spec1 is invalid.

Fig. 2. Incorrect submission to `spec1` in Alloy4Fun

relates each photo to exactly **one** day when it was posted; advertisements are a particular kind of photo, introduced by sub-signature `Ad`.

When validating a system design, one would impose additional restrictions over this model using temporal relational logic through *facts*. To promote maintainability, reusable formulas and expressions can be introduced through *predicates* and *functions*, respectively. Then *run* and *check* commands would be defined to animate the model or verify desirable properties, respectively. Commands are automatically executed by the Alloy Analyzer within a given bound for the universe. When teaching Alloy, a typical kind of challenge presented to students is to encode some of these logical constraints.

With this in mind, Alloy4Fun introduced the concept of model *secret*, allowing such challenges to be auto-graded [16]. Instructors write an oracle as a secret predicate and then use the Analyzer to check whether a student submission is equivalent to it. Two examples are shown in the bottom of Fig. 1. The student is asked to write in predicate `spec1` the constraint *"every photo is posted by one user"*. Hidden from the student through annotation *//SECRET*, predicate `oracle1` specifies a possible solution: for every photo `p`, there is exactly one user related with it through `posts`. Command `spec1` simply checks whether the student specification and the oracle are equivalent (with at most 3 atoms in each signature). Being a semantic test, the correct submission can be syntactically different from the oracle. A single Alloy4Fun model (which we call an *exercise*) can contain multiple *challenges*; the one in Fig. 1 has 2.

If a check command is invalid, the Analyzer (and Alloy4Fun) returns a graph-shaped counter-example where the equivalence does not hold. The user can navigate through alternative counter-examples and customize the visualization for better comprehension. As an example, Fig. 2 shows the student view of the exercise from Fig. 1 (i.e., secrets are hidden), where the student submitted an incorrect attempt to the `spec1` challenge and a counter-example was returned. In principle, counter-examples are helpful when debugging specifications, but studies show they are not the most adequate feedback for novice users [6].

Alloy4Fun collects anonymous data from all user interactions. So, whenever a student runs a command, it stores information such as the full model, the selected command and its outcome, and the identifier of the model it derived from. The resulting derivation tree allows the reconstruction of student paths, by identifying sequential attempts to the same challenge. The already mentioned dataset [15] collects this data for various editions of formal methods courses in the Universities of Minho and Porto, Portugal, between the Fall of 2019 and the Spring of 2023, totalling about 100 000 models.

3 Automatic Hint Generation

Next-step hints Although next-step hints are a popular kind of feedback in ITSs, there are some concerns that such hints may be counter-productive, namely due to hint abuse and avoidance [1], or the fact that they indicate students 'how' to fix rather than 'why' [18]. Nonetheless, studies [10,14,25,26] suggest that next-step hints have no impact on long-term learning retention but often improve immediate performance, enabling students to learn more efficiently. A recent study on Alloy reached similar conclusions [6]. Moreover, there's an indication that accompanied by prompts for self-explanation, such hints may improve learning retention [20], although the results could not be replicated [19].

There are several techniques to automatically generate a next-step hint from an incorrect submission [21]: searching for steps that take the student closer to a reference solution, using previous successful submissions by peers, identifying known patterns in the incorrect submission, or trying to repair a solution to pass an oracle. Repair-based approaches have been proposed for Alloy, which we discuss below. However, these are often affected by scalability issues, and it's unclear how to select high-quality hints from alternative repair suggestions. In contrast, data-driven approaches do not suffer from performance issues and may generate more intuitive hints since they are based on historical submissions. The tradeoff is that they may be ineffective in large solution spaces or assignments with small historical logs. We are not aware of such techniques for specification ITSs, so we discuss them in the context of programming ITSs next.

Data-driven hint generation. The first data-driven hint generation approach was proposed in the context of a logic-proof tutoring system [2]. It has since been adapted to platforms for programming [11,23,28], although not for specifications, as far as we are aware. The main idea behind these approaches is to use historical student submissions to build a graph of all traversed solution paths. Each node in the graph is the AST of a submitted attempt in a student path, and the transitions register the sequence of edit actions that lead from one submission to the other. To build the hint graph, all student paths are combined into a single graph by matching identical submissions, keeping the popularity of each state and/or transition, and marking correct submissions as goal states. When a student asks for a hint, if the current state is present in the hint graph, it calculates the optimal path towards a correct solution and generates a hint. In [2] Markov Decision Processes (MDP) were used to calculate the optimal path, but various other policies have since been proposed [22,24]. Studies have used expert input to evaluate the quality of the hints resulting from different polices [22,24].

The main challenge for this kind of approach is the size of the solution space. Besides being an obvious issue for assignments with little historical data, the solution space for expressive programming languages is so large that getting hits in the graph may be unlikely even with substantial historical data. Several approaches have been explored to address this, such as creating intermediate states [28], using program outputs rather than the actual AST as graph states [11], or employing canonicalization techniques to group semantically equivalent ASTs in the same graph state [27].

Automated Alloy Repair. Automated program repair techniques generate fixes for programs that fail to pass a certain oracle. In education, this oracle can be written by the instructor, either a reference solution or a suite of tests, and then used to generate hints to fix student submissions. Some automated repair techniques have been proposed for Alloy specifications [3,4,30,31].

ARepair [30] was the first repair technique for Alloy, using test cases as oracle. This makes it prone to overfitting, generating fixes that pass the tests but still break the expected properties. Moreover, Alloy models are typically not accompanied by test cases. In contrast, BeAFix [3] uses as oracles check commands. This is more natural in Alloy (and Alloy4Fun challenges) since models are typically accompanied by commands defining expected properties. Unfortunately, the pruning techniques proposed to improve performance rely on multiple commands and suspicious locations, and are not effective for simple Alloy4Fun specification challenges. TAR [4] was developed for the educational context and integrated into Alloy4Fun. It is focused on producing timely feedback to avoid student frustration (and to support the temporal aspects of Alloy 6). Its pruning technique evaluates previously seen counter-examples to avoid costly calls to the solver. It was shown to considerably outperform ARepair and BeAFix within a 1-minute timeout, but it is unfeasible for specifications far from a correct solution. ATR [31] is another technique to repair Alloy 4 specifications with commands as oracles. Although developed independently from TAR, it also uses counter-examples (and the closest valid instances) to avoid calls to the Analyzer. ATR

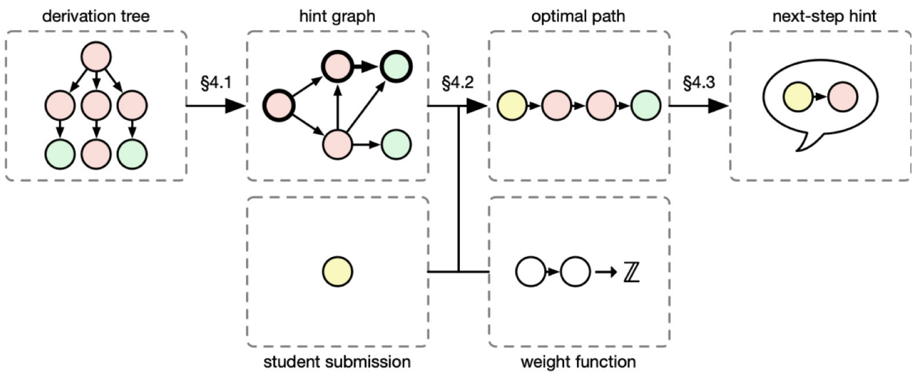


Fig. 3. Overview of the approach when submissions are present in historical data

was shown to outperform the repair rate of ARepair and BeAFix, and to be more efficient than BeAFix.

4 Hints from Historical Alloy Data

The proposed technique adapts existing data-driven hint generation techniques for programming. Using Alloy4Fun historical data, it creates a graph that captures students' progress when solving a challenge, which is then used to generate hints for future students. This section describes the technique and its implementation, whose overview is presented in Fig. 3.

4.1 Hint Graph Construction

To generate hints, our approach relies on a graph of student submissions for each specification challenge, created from an Alloy4Fun dataset. These graphs are created offline and can be rebuilt from time to time as new data is collected. Each node in the graph is a normalized formula previously submitted by a student, labelled as correct or incorrect, and each edge represents a transition between two submissions. Each formula is unique in the graph, so similar submissions are merged, and the frequency of nodes and transitions are registered to be used in the pathfinding step. Formula comparison is performed at the AST level, so syntactically incorrect entries in the dataset are disregarded. As seen in Sect. 2, an Alloy4Fun exercise may contain multiple challenges, so the derivation tree must be split per challenge. The Alloy command called by each entry identifies the corresponding target challenge. To exactly identify the student submission and avoid considering the oracle as part of the graph state, we assume that each challenge command calls an empty predicate to be filled by the student, as exemplified in Fig. 1; the formula for each node is extracted from the content of

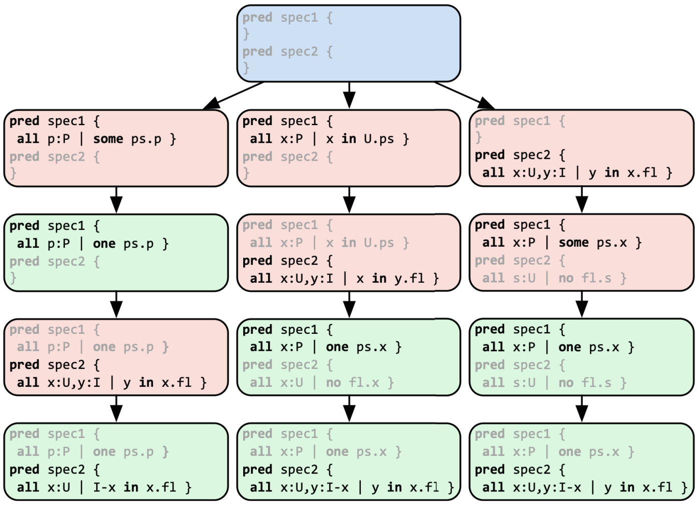


Fig. 4. A sample derivation tree with 3 paths for the exercise in Fig. 1

that predicate[2]. When extracting submissions to a certain challenge and removing syntactically invalid formulas, the pointer to the parent submissions must be updated accordingly to preserve the student paths.

For improved efficacy (i.e., the probability of a submission having a match in the graph), we apply a few canonicalizations specified in [27] that were sensible in the Alloy context, such as *sorting commutative operations* and *normalizing the direction of comparisons*. Additionally, since quantified variables in Alloy cannot be inlined, we apply *variable anonymization*. The same transformation is applied to submissions whenever a hint is requested. Note that we do not want to abuse canonicalization and end up with hints for a formula that differs too much from the concrete student submission. So, for example, we do not propagate **not** operators using De Morgan's laws.

To illustrate this process, consider the derivation tree in Fig. 4, that could be collected from the exercise in Fig. 1 (signature and field names abbreviated). It contains 3 paths, with incorrect and correct interleaved attempts to both challenges (spec1 and spec2). The target challenge in each state is the one not greyed-out, green and red nodes represent correct and incorrect submissions, respectively, and the blue node is the root model shared by the instructor[3]. This will result in the two graphs in Fig. 5, with node and transition frequency identified by line thickness. Notice the normalization before merging, here just the name of the quantified variables. Notice also that there may be more than one semantically equivalent valid solution per challenge.

[2] This strategy may not hold for other kinds of Alloy4Fun challenges, in which case additional annotations could be used to identify the submission predicate.

[3] Technically, paths can branch if a student backtracks to a previous model. This phenomenon was negligible in the dataset, and does not affect the general procedure.

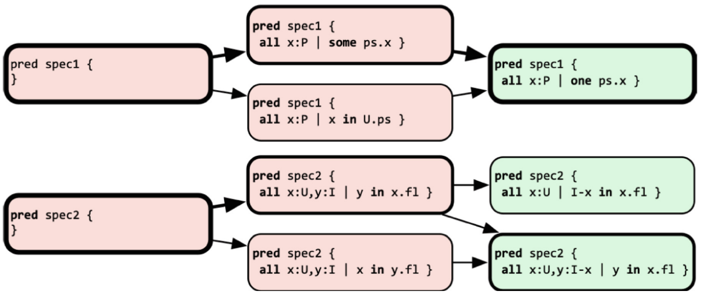


Fig. 5. Hint graphs resulting from the derivations in Fig. 4

4.2 Finding the Optimal Next State

The hint generation algorithm runs on demand when a student requests a hint. After locating the student's submission in the hint graph of the target challenge, the *current state*, the algorithm searches for the optimal path—according to the defined criterion—from it to any correct formula, the *goal state*. The first edge of this path indicates the transition the student should make to progress toward the goal, the *next state* that will be used to create the hint.

As discussed in Sect. 3, several criteria have been proposed to define the optimal path. Our goal was to keep the path finding process as general as possible, so we allow the instructor to define the desirable policy. This is done through the definition of a weight function on the edges of the graph from a set of available attributes. These attributes may be data-driven—namely the (relative) popularity of the edge in the source state, and the popularity of the source and target states—but also syntactic—namely the complexity of the edge transformation and the source and target formulas. The complexity of the states is given by the size of the respective AST. For the complexity of the edge, recall that a transition between states may encompass several actions between two successive submissions from the student. We measure the complexity of the edge as the tree edit distance (TED) between the two states, calculated using the state-of-the-art algorithm APTED[4].

Given the weight function on edges, the optimal path is calculated through a simple shortest path algorithm for weighted graphs.

4.3 Hint Message Generation

The next-step hint is generated from the optimal path. We consider two aspects to create the hint message: how far the student is from the optimal solution, based on the TED between the current and the goal states; and the sequence of edit operations between the current and the next states. To calculate this sequence, we use an implementation[5] of GumTree [9], which calculates a mapping

[4] https://github.com/DatabaseGroup/apted.
[5] https://github.com/GumTreeDiff/gumtree.

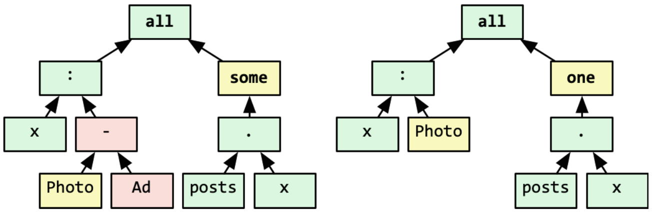


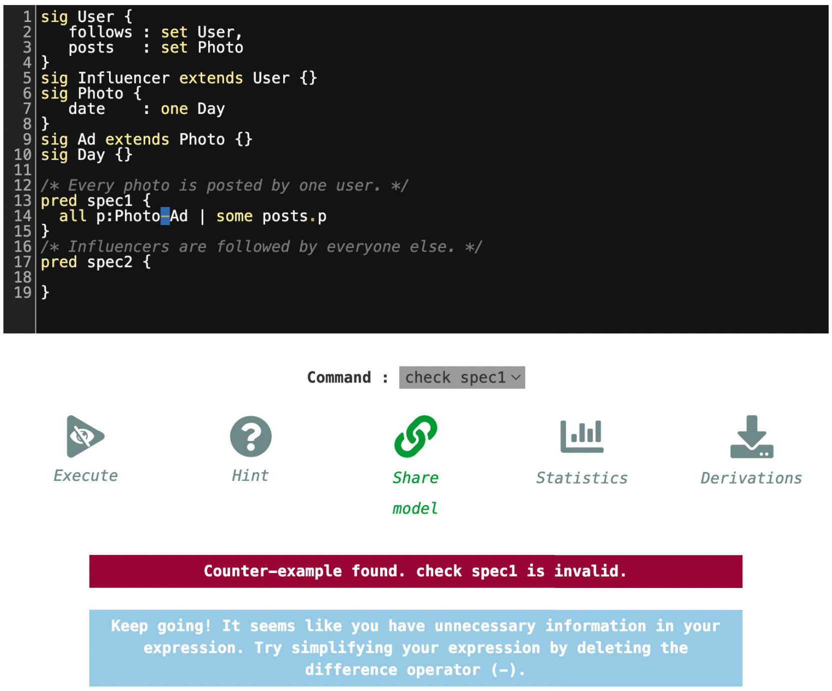
Fig. 6. Example of AST edit operations.

between AST nodes and uses the Chawathe et al. [5] algorithm for computing the edit sequence. The result is a sequence of *inserting*, *deleting*, or *moving* nodes, or *updating* a node's label. Since there may be dependencies between these edit operations, currently we select the first operation of the sequence for the hint. To translate an edit operation to a hint we use a message template for each operation type. The messages try to simulate what a teacher would say to a struggling student, and contain placeholders for operator-specific information that can be tailored for the Alloy language.

Consider, e.g., transforming `all p:Photo-Ad | some posts.p`, incorrect for `spec1`, into the correct `all p:Photo | one posts.p`, shown in Fig. 6. This requires 4 operations: move node `Photo` up, delete nodes `-` and `Ad`, and update node `some` to `one`, resulting in a TED of 4. The resulting hint message looks like this: *"Keep going! It seems like you have unnecessary information in your expression. Try simplifying your expression by deleting the difference operator (-)."*.

4.4 Handling Missing Hits

A pure data-driven approach fails for formulas absent from the historical data. To improve efficacy, one can construct paths from a previously unseen state until one already in the graph. To this purpose, we enhance our data-driven approach with a mutation-based component. Whenever a request does not exist in the graph, we generate variants according to a set of mutators. If a variant happens to already exist in the graph, a temporary edge from the current state to that variant is added with popularity 0, thus connecting the previously unseen formula to the graph and enabling the pathfinding procedure. These mutators—which are comprised by multiple edit actions—represent typical high-level transformations applied to a formula. In particular, we rely on the mutators proposed by TAR [4], which were specifically designed for the Alloy language. Currently, this process is restricted to a single mutation to avoid reaching a path too distinct from the student submission.



```
sig User {
   follows : set User,
   posts   : set Photo
}
sig Influencer extends User {}
sig Photo {
   date    : one Day
}
sig Ad extends Photo {}
sig Day {}

/* Every photo is posted by one user. */
pred spec1 {
  all p:Photo-Ad | some posts.p
}
/* Influencers are followed by everyone else. */
pred spec2 {

}
```

Command : check spec1

Counter-example found. check spec1 is invalid.

Keep going! It seems like you have unnecessary information in your expression. Try simplifying your expression by deleting the difference operator (-).

Fig. 7. Hint provided for incorrect submission to `spec1` in extended Alloy4Fun

4.5 Deployment in Alloy4Fun

The proposed approach was implemented as a REST service, and we implemented an extension to the Alloy4Fun platform that uses the service to automatically provide hints to challenge attempts[6]. A new button was added to the interface that allows users to request a hint when an incorrect specification is submitted to a challenge. If the tool is able to generate a hint, it highlights a location in the editor and provides an explanatory message. This is shown in Fig. 7 for the example used in Sect. 4.3.

The service was implemented in Java—to take advantage of the Alloy Analyzer parser and AST—using the Quarkus framework. The hint graphs are stored in a new collection for the MongoDB database of Alloy4Fun. The weight function that determines the policy is provided through a JSON file that defines an arithmetic expression over the complexity and frequency attributes presented in Sect. 4.2.

Although optimal paths could be calculated live from the graph whenever a hint is requested, in practice, to make hint generation as fast as possible, we pre-compute the optimal next state for every state of the graph offline. When a hint is requested, it is just a matter of fetching the next state from the graph.

[6] https://github.com/anaines14/Alloy4Fun.

Table 1. Statistics for the considered exercises

Exercise	Id	Challs.	Specs.	Syntatic	Training	Testing
Social Network	SN	8	22690	14943	10428	2793
Courses	Co	15	22516	14911	10431	2418
Train Station	TS	10	8158	6388	4394	1331
Production Line	PL	10	8078	6058	4156	1102
Trash LTL	TL	19	5279	4352	2788	890
Classroom FOL	CF	14	5893	4376	2702	663
Classroom RL	CR	14	6341	4248	2474	687
Trash RL	TR	9	4361	3059	1530	347
Trash FOL	TF	9	4092	2719	1425	194
Graphs	Gr	7	3211	2481	1281	370
Labelled Transition System	TS	6	3382	2076	995	393
Curriculum Vitae	CV	4	1199	854	596	218

Table 2. Quantitative evaluation results, all times in seconds

Id	Construction			Data-driven		Data+Mutations		TAR		
	States	T_G	T_P	Hits	T_H	Hits	T_H	Hits	T_H	Cmn.
SN	3605	165.5	7.9	1265 (45%)	0.01	1740 (62%)	0.7	539 (19%)	44.6	20
Co	4104	215.3	13.2	812 (34%)	0.01	1298 (54%)	0.7	502 (21%)	38.6	30
TS	1874	69.8	9.5	529 (40%)	0.02	751 (56%)	0.8	320 (24%)	38.2	9
PL	1862	113.6	5.4	373 (34%)	0.01	525 (48%)	0.7	297 (27%)	38.5	15
TL	1219	52.2	7.7	357 (40%)	0.01	569 (64%)	0.4	771 (87%)	9.5	23
CF	903	45.2	6.0	331 (50%)	0.01	463 (70%)	0.5	182 (27%)	26.7	13
CR	985	46.7	5.5	239 (35%)	0.02	330 (48%)	0.3	159 (23%)	35.9	7
TR	446	31.5	2.4	195 (56%)	0.02	260 (75%)	0.2	237 (68%)	0.9	3
TF	343	28.6	1.9	100 (52%)	0.02	154 (79%)	0.3	167 (86%)	0.9	5
Gr	599	27.7	3.7	162 (44%)	0.02	251 (68%)	0.2	175 (47%)	21.0	0
TS	489	22.8	3.7	74 (19%)	0.02	141 (36%)	0.2	37 (9%)	8.2	0
CV	324	11.7	1.6	44 (20%)	0.01	48 (22%)	0.7	61 (28%)	48.2	2

Table 3. Incorrect specifications selected for the questionnaire

Spec	Id	Incorrect
spec1	I1a	**all** p:Photo | **some** u:User | u→p **in** posts
	I1b	**all** p:Photo | p **in** User.posts
	I1c	**all** p:Photo,u:User | p **in** u.posts
spec2	I2a	**all** i:Influencer,u:User | i **in** u.follows
	I2b	**all** u:User | Influencer **in** u.follows
	I2c	**all** u:User | u.follows **in** Influencer

5 Evaluation

We evaluate the proposed hint generation technique quantitatively—addressing its effectiveness and efficiency—and qualitatively—comparing the generated hints with those suggested by experts. Specifically, we aim to answer the following research questions:

RQ1 How effective is the tool when a hint is requested, i.e., how often can it generate a hint?
RQ2 How efficient is it in the various steps of the process, i.e., how long does it take to construct the graph and to generate a hint?
RQ3 How does it compare with repair-based approaches?
RQ4 What is the quality of the generated hints, and what is the impact of the specified policy?

Table 4. Most popular answers by expert Alloy tutors

Id	Most popular location	#	Most popular hint	#
I1a	**some** u:User | …	8	Update @ **some** u:User | …	8
I1b	**all** p:Photo | …	3	Update @ … **in** …	2
			Update @ **all** p:Photo | …	2
I1c	**all** p:Photo,u:User | …	7	Update @ **all** p:Photo,u:User | …	5
I2a	**all** i:Influencer,u:User | …	4	Delete @ u:User	2
			Update @ **all** i:Influencer,u:User | …	2
			Insert @ **all** i:Influencer,u:User | …	2
I2b	Influencer	5	Insert @ Influencer	4
I2c	… **in** …	6	Update @ … **in** …	5

5.1 Quantitative Evaluation

For the quantitative evaluation, we applied our technique to the Alloy4Fun dataset [15], which contains data for multiple exercises (each with multiple challenges). It contains about 66 000 syntactically correct student submissions to 12 different exercises, collected over 4 years. Table 1 shows the number of challenges per exercise (Challs.) and the aggregated statistics. The dataset was split into a training subset to construct the graphs and a testing subset to evaluate the performance. We split full paths in the dataset randomly 70%/30% (rather than splitting individual submissions, since our approach is based on previously traversed paths). Each entry in the testing subset was then run for a hint request in the purely data-driven technique, in the version that employs mutations for formulas absent in the graph, and also in the existing repair-based approach TAR with a maximum search depth of 2. All tests were performed on a commodity Intel Core i5-13600KF, with 32 GB of RAM. Timeout for requests was set to 1 min since timely feedback is critical in the educational context. Table 2 summarizes the results.

Regarding **RQ1**, Table 2 shows the hit rate (i.e., the number of specifications for which the tool was able to return a hint) for the purely data-driven and the mutation-enhanced versions. The hit rate of the former ranges from 19% to 56%, with a total average of 39%. Interestingly, the exercises with higher hit rate are not among those with the largest number of specifications in the historical log, which is possibly connected to the complexity of the challenges. Nonetheless, this hit rate will only increase as the exercises collect more submissions. Activating the mutation component for missed requests considerably increases the hit rate to an average of 57%.

For **RQ2**, we start with the graph construction step. Table 2 aggregates the results for each exercise, namely the number of unique formulas resulting in graph states, and the time to construct the graphs (T_G) and to compute the optimal next state (T_P). The selected weight function did not affect the performance significantly (shown values are for minimizing transition complexity). Results show that the whole process takes a few minutes for the exercises with more submissions, which is reasonable since this construction is expected to be performed sporadically offline. Regarding the hint generation step, Table 2 also shows the average time to generate a hint for both approaches (T_H). For the data-driven approach, this time is negligible for all exercises (recall that we precalculate the optimal next state offline). When enhanced with mutations, there is an expected increase on time, although still below 1 s in average. This makes the technique feasible in answering live hint requests.

Regarding **RQ3**, Table 2 also shows the hit rate and time to retrieve a hint for TAR. The hit rate seems less predictable, ranging from 9% to 87%, with an average of 30%, well below our approach. Interestingly, the number of formulas for which both our data-driven approach and TAR can generate hints (Cmn.) is very small, suggesting that these approaches are complementary. As expected TAR takes considerably longer to generate a hint, with an average of 27 s, since it is search-based and calls the solver to validate potential solutions.

5.2 Qualitative Evaluation

To evaluate the quality of the generated hints (**RQ4**), we asked experienced Alloy instructors how they would suggest a next-step hint for a set of incorrect specifications. For each of the two challenges from Fig. 1, we selected 3 frequently submitted incorrect specifications, shown in Table 3. We created a questionnaire that asked for hints in the shape of a target location and an edit operation (insertion, removal and update). We sent the questionnaire to 12 Alloy instructors unrelated with this work, and received 8 replies. We observed that, except for one case (`I1a`), the experts did not select the same next-step hint, highlighting the difficulty of automatically generating hints. Table 4 shows the most popular answers by the experts, both by location only by the whole hint (i.e., location plus edit operation).

Our approach allows policies to be customized through weight functions. To compare the answers of the experts with the results of our approach, we designed a few simple weight functions, some considering only the complexity of nodes (Cmp_N) and edges (Cmp_E), and others only the frequency of nodes (Frq_N) and edges (Frq_E). We also considered a couple of policies that combined these syntactic and data-driven attributes. For this evaluation, we do not consider the mutation-enhanced version of the technique, as we intend to evaluate the quality of the data-driven approach. For each policy we counted in how many of the 6 incorrect specifications the generated hint: *i*) was selected by *any* expert, and *ii*) was among the most *pop*ular answers by the experts. We consider whether there was a match only on the identified location or in the whole hint. Table 5 shows the results.

Interestingly, results show that looking uniquely at the complexity of the edges (TED) results in hints closer to the experts than the purely data-driven policies. However, the best results are actually when considering both kinds of attributes simultaneously: with Cmp_E and Frq_E every hint generated was one also suggested by some experts, and often one of the most popular.

Table 5. Matches between hints generated by policies and expert hints

Policy	Location		Loc.+Op.	
	Any	Pop.	Any	Pop.
Cmp_N	3	3	3	3
Frq_N	2	2	2	2
Cmp_E	5	3	4	2
Frq_E	4	3	4	2
$Cmp_N \times Frq_E$	6	4	5	3
$Cmp_E \times Frq_E$	6	5	6	3

6 Conclusion

This paper presented the first data-driven hint generation technique for ITSs for learning formal specifications, namely for the Alloy language, and its implementation in the Alloy4Fun platform. The data-driven technique is complemented with a mutation-based component to handle absences in the historical data. Our evaluation shows that our approach outperforms an existing repair-based technique, and that with the right policy the generated hints can emulate those provided by experts.

Our expert questionnaires included an open question where most experts suggested feedback in shapes other than next-step hints, such as explaining the issue with the incorrect specification. Some studies suggest next-step hints accompanied by self-explanations can improve learning [20], but studies also find hints explaining issues are not well-received by novices [6]. Further studies are needed on how to implement these effectively. On the other hand, the quantitative evaluation showed a small overlap between the cases successfully handled by the data-driven and the repair-based approaches, suggesting that hybrid approaches may be worth exploring.

Acknowledgments. The work by A. Barros and H. Neto is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project EXPL/CCI-COM/1637/2021. The work by N. Macedo is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aleven, V., Roll, I., McLaren, B.M., Koedinger, K.R.: Help helps, but only so much: Research on help seeking with intelligent tutoring systems. Int. J. Artif. Intell. Educ. **26**(1), 205–223 (2016)
2. Barnes, T., Stamper, J.C.: Toward automatic hint generation for logic proof tutoring using historical student data. In: ITS, LNCS, vol. 5091, pp. 373–382. Springer (2008). https://doi.org/10.1007/978-3-540-69132-7_41
3. Brida, S.G., et al.: Bounded exhaustive search of Alloy specification repairs. In: ICSE, pp. 1135–1147. IEEE (2021)
4. Cerqueira, J., Cunha, A., Macedo, N.: Timely specification repair for Alloy 6. In: SEFM, LNCS, vol. 13550, pp. 288–303. Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_18
5. Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: SIGMOD Conference, pp. 493–504. ACM Press (1996)
6. Cunha, A., Macedo, N., Campos, J.C., Margolis, I., Sousa, E.: Assessing the impact of hints in learning formal specification. In: SEET@ICSE, pp. 151–161. ACM (2024)

7. Danas, N., Nelson, T., Harrison, L., Krishnamurthi, S., Dougherty, D.J.: User studies of principled model finder output. In: SEFM, LNCS, vol. 10469, pp. 168–184. Springer (2017). https://doi.org/10.1007/978-3-319-66197-1_11
8. Dyer, T., Nelson, T., Fisler, K., Krishnamurthi, S.: Applying cognitive principles to model-finding output: the positive value of negative information. Proc. ACM Program. Lang. **6**(OOPSLA1), 1–29 (2022)
9. Falleri, J., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: ASE, pp. 313–324. ACM (2014)
10. Gusukuma, L., Bart, A.C., Kafura, D.G., Ernst, J.: Misconception-driven feedback: results from an experimental study. In: ICER, pp. 160–168. ACM (2018)
11. Hicks, A., Peddycord III, B.W., Barnes, T.: Building games to learn from their players: generating hints in a serious game. In: ITS, LNCS, vol. 8474, pp. 312–317. Springer (2014). https://doi.org/10.1007/978-3-319-07221-0_39
12. Jackson, D.: Software abstractions: Logic, language, and analysis. MIT Press, revised edn. (2006)
13. Jackson, D.: Alloy: a language and tool for exploring software designs. Commun. ACM **62**(9), 66–76 (2019)
14. Lazar, T., Sadikov, A., Bratko, I.: Rewrite rules for debugging student programs in programming tutors. IEEE Trans. Learn. Technol. **11**(4), 429–440 (2018)
15. Macedo, N., Cunha, A., Paiva, A.C.R.: Alloy4Fun dataset for 2022/23 (2023). https://doi.org/10.5281/zenodo.8123547, https://doi.org/10.5281/zenodo.8123547
16. Macedo, N., et al.: Experiences on teaching Alloy with an automated assessment platform. Sci. Comput. Program. **211**, 102690 (2021)
17. Mansoor, N., Bagheri, H., Kang, E., Sharif, B.: An empirical study assessing software modeling in Alloy. In: FormaliSE. pp. 44–54. IEEE (2023)
18. Marwan, S., Lytle, N., Williams, J.J., Price, T.W.: The impact of adding textual explanations to next-step hints in a novice programming environment. In: ITiCSE, pp. 520–526. ACM (2019)
19. Marwan, S., Price, T.W.: iSnap: evolution and evaluation of a data-driven hint system for block-based programming. IEEE Trans. Learn. Technol. **16**(3), 399–413 (2023)
20. Marwan, S., Williams, J.J., Price, T.W.: An evaluation of the impact of automated programming hints on performance and learning. In: ICER, pp. 61–70. ACM (2019)
21. McBroom, J., Koprinska, I., Yacef, K.: A survey of automated programming hint generation: the hints framework. ACM Comput. Surv. **54**(8), 1–27 (2022)
22. Piech, C., Sahami, M., Huang, J., Guibas, L.J.: Autonomously generating hints by inferring problem solving policies. In: L@S, pp. 195–204. ACM (2015)
23. Price, T.W., Dong, Y., Barnes, T.: Generating data-driven hints for open-ended programming. In: EDM, pp. 191–198. Int. Educ. Data Min. Soc. (IEDMS) (2016)
24. Price, T.W., et al.: A comparison of the quality of data-driven programming hint generation algorithms. Int. J. Artif. Intell. Educ. **29**(3), 368–395 (2019)
25. Price, T.W., Marwan, S., Winters, M., Williams, J.J.: An evaluation of data-driven programming hints in a classroom setting. In: AIED (2), LNCS, vol. 12164, pp. 246–251. Springer (2020)
26. Rivers, K.: Automated Data-Driven Hint Generation for Learning Programming. Ph.D. thesis, Carnegie Mellon University, USA (2017)
27. Rivers, K., Koedinger, K.R.: A canonicalizing model for building programming tutors. In: ITS, LNCS, vol. 7315, pp. 591–593. Springer (2012)

28. Rivers, K., Koedinger, K.R.: Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. Int. J. Artif. Intell. Educ. **27**(1), 37–64 (2017)
29. Sarkar, A., Negreanu, C., Zorn, B., Ragavan, S.S., Pölitz, C., Gordon, A.D.: What is it like to program with artificial intelligence? In: PPIG, pp. 127–153. Psychology of Programming Interest Group (2022)
30. Wang, K., Sullivan, A., Khurshid, S.: Automated model repair for Alloy. In: ASE, pp. 577–588. ACM (2018)
31. Zheng, G., et al.: ATR: template-based repair for Alloy specifications. In: ISSTA, pp. 666–677. ACM (2022)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

B2SAT: A Bare-Metal Reduction of B to SAT

Michael Leuschel(✉)

Faculty of Mathematics and Natural Science,
Heinrich Heine University Düsseldorf, Düsseldorf, Germany
michael.leuschel@hhu.de

Abstract. We present a new SAT backend for the B-Method to enable new applications of formal methods. The new backend interleaves low-level SAT solving with high-level constraint solving. It provides a "bare metal" access to SAT solving, while pre- and post-calculations can be done in the full B language, with access to higher-order or even infinite data values. The backend is integrated into ProB, not as a general purpose backend, but as a dedicated backend for solving hard constraint satisfaction and optimisation problems on complex data. In the article we present the approach, its origin in the proof of Cook's theorem, and illustrate and evaluate it on a few novel applications of formal methods, ranging from biology to railway applications.

1 Introduction

The B-method [1,2] is based on predicate logic, arithmetic and set theory. Particularly when using Unicode symbols, its core syntax is very close to standard mathematical notation. The ProB validation tool [29] for B can bring this mathematics to life [28]. We have been using this fact to develop a variety of interactive teaching materials for an undergraduate theoretical computer science course, in particular via Jupyter notebooks [15]. In many cases the mathematical formulas in the course script [36] are valid B formulas or need only minor adaptations. These notebooks[1] cover topics like finite automata, automata determinatisation and minimisation, parsing algorithms, Turing machines, various Gödelisations and conversions of grammars to automata models and back.

In the summer of 2023 we have covered for the first time Cook's theorem [8] in more detail and developed an accompanying B model for it. Cook's theorem states that SAT solving (satisfiability of propositional logic formulas) is NP-complete. The proof shows that a successful run of a non-deterministic Turing machine (solving a given NP-problem) can be modelled as the solution of a propositional logic formula. The translation rules to SAT in [8] and [36] are written as quantified logic formulas (using universal and existential quantification over time points, tape contents and states of the Turing machine). These

[1] Some of the notebooks are available at: https://gitlab.cs.uni-duesseldorf.de/general/stups/prob-teaching-notebooks.

© The Author(s) 2025

formulas can be encoded elegantly in B, and we could take them almost verbatim from the script of the lecture [36] to develop a B encoding of the proof.[2] As we will see in Sect. 2, one can thus visualise solutions of the translated SAT problem, giving students a better intuition about Cook's theorem. In the process of formalisation, we also found a few subtle mistakes in the script [36].

More importantly, however, is the realisation that this mathematical style of describing a SAT problem is useful for other, new applications of formal methods we were working on (from biology, data mining and railways). This led to the development of the new solving backend B2SAT. B2SAT intertwines the solving of high-level, higher order B predicates with solving "bare-metal" SAT problems. When applied to the B encoding of Cook's theorem it creates and solves exactly the SAT problem described in [8,36] and translates the SAT solution back to B.

In the rest of the paper we first show more details about Cook's theorem and how to describe the underlying SAT problems in predicate logic. We then describe the technique and implementation of our new solving backend B2SAT. We show that B2SAT has applications for complex constraint solving and optimisation tasks, enabling convenient modelling in B, TLA$^+$ or Z and effective solving in SAT. For some applications at least, it is considerably more efficient than existing solvers for B, TLA$^+$ or Z.

2 Cook's Theorem in B

Cook's theorem [8] states that SAT solving is NP-complete. The theorem is still important 50 years after, see, e.g., [34].

Recall that a set A of words is in NP if there is a non-deterministic Turing machine M_A which accepts A in polynomial time. The proof of NP-hardness in [8] shows that accepting computations of M_A for a particular input x correspond to solutions of a propositional logic formula F_x. The formula F_x is derived from x and the Turing machine M_A. Nowadays we would call this process *bounded model checking of the Turing machine* M_A. Indeed, the number of computation steps of the Turing machine is bounded by some polynomial π_A over the size of the input. As such, the reachable cells of the Turing machine's infinite tape are also bounded for a given input x, making it possible to encode the whole computation as a propositional logic formula.

We now present a B encoding of the core of the proof, based on encoding the translation rules deriving the SAT formula F_x from M_A and the input x. We here assume that M_A is a Turing machine with one tape. The translation rules to SAT are represented as predicate logic formulas; these formulas can be encoded elegantly in B, and are taken almost verbatim from [36].

A Turing machine consists of a working alphabet Γ, containing the input alphabet Σ and the blank symbol, a set of states S, an initial state z_0, and a set $F \subseteq S$ of final states. In addition we have the transition relation $\delta \in (S \times \Gamma) \leftrightarrow (S \times \Gamma \times Dir)$, where $Dir = \{L, R, N\}$ is the set of possible head movements

[2] The major adaptations were changing $\bigwedge_{i \in S} \phi(i)$ to $\forall i.i \in S \Rightarrow \phi(i)$ and $\bigvee_{i \in S} \phi(i)$ to $\exists i.i \in S \wedge \phi(i)$ and expressing "exactly-one" constraints as cardinality constraints.

(left, right, neutral). Here, $(z, y) \mapsto (z', y', d) \in \delta$ means that if the machine is in state $z \in S$ and the tape contains the symbol $y \in \Gamma$ at the head position, then the machine can change its state to z', write y' at the current tape position (overwriting y) and move its head according to d. Note that we model δ here as a relation, and the Turing machine can be non-deterministic.

We now assume that M accepts a set $A \in NP$ (i.e., a problem in NP). Hence the number of steps for accepting an input $x \in A$ is bounded by some polynomial π. Hence, we model the set of time points $Time = 0 \mathrel{..} \pi(n)$ where n is the size of the input x. In that time span the Turing machine can move only a bounded number of steps left or right, hence we have a set of reachable tape positions $Pos = -\pi(n) \mathrel{..} \pi(n)$

We can specify the set of valid computation paths of the Turing machine which accept x by the formula F_x. The propositional logic variables of F_x depend on the size of the input x and are modelled as these three total functions in B:

- $state \in Time \times States \rightarrow BOOL$, encoding the state of the Turing machine at each time point $t \in Time$,
- $head \in Time \times Pos \rightarrow BOOL$, encoding the position of the head on the tape at each time point,
- $tape \in Time \times Pos \times \Sigma \rightarrow BOOL$, encoding the contents of the tape at each time point.

The formula F_x is a conjunction $S \wedge K \wedge U1 \wedge U2 \wedge E$ of the following five subformulas, which we partially describe below:

S The initial condition specifying that $head(0, 0) = TRUE$, $state(0, z0) = TRUE$ and the intial tape contents.

K A correctness condition stating that the Turing machine
  - can only be in a single state: $\forall t.(t \in Time \Rightarrow card(\{s \mid state(t, s) = TRUE\}) = 1)$,
  - have a single head position and symbol at every tape position.

U1 Updating the state, head position and the tape's contents at the head.

U2 the Frame axiom, stating that tape contents not at the head remain unchanged: $\forall(t, i, a).(t \in Time^{+} \wedge i \in Pos \wedge a \in \Gamma) \Rightarrow ((head(t, i) = FALSE \wedge tape(t, i, a) = TRUE) \Rightarrow tape(t+1, i, a) = TRUE)))$ where $Time^{+}$ is all but the last time point.

E ensuring we reach a final state: $\exists s.(s \in Final \wedge state(\pi(n), s) = TRUE)$.

Parts of the B model can also be found in Fig. 2. U1 is the most complicated formula, and was not written out in [8]. U1 contained four mistakes in [36], which were undetected for at least 15 years. E.g., [36] did not ensure that we do not step outside the set of modelled positions.

We can now bring this formula to life by letting ProB find solutions for *state*, *head* and *tape* which satisfy F_x for a particular Turing machine and a particular input. In Fig. 1 we show the graphical rendering[3] of one such solution

[3] The B model and an HTML export of this visualisation can be inspected at https://stups.hhu-hosting.de/models/B2SAT.

for a Turing machine with 4 states[4] the input "100" (1 is green, 0 is red and BLANK is white) and modelling 16 computation steps (from left to right).

The B model can now be used to show the students the importance of the various subformulas of F_x. For example, Fig. 1 on the right shows what happens when we drop the frame axiom U2, meaning that untouched tape contents can change willy-nilly at each step.

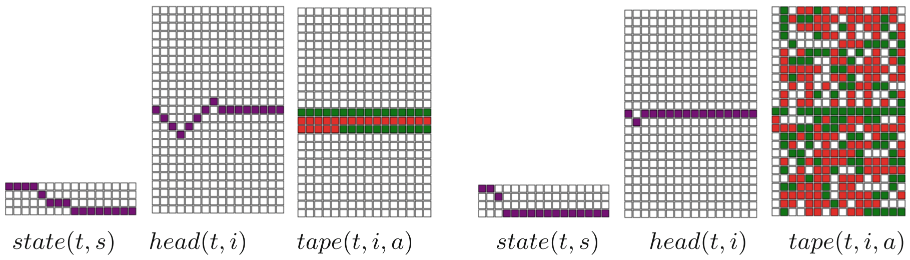


Fig. 1. Two solutions of F_x, showing SAT variables for state, head and a condensed view of tape. Time t progresses from left to right in each case. The SAT problems have 1664 variables. For the right the frame axiom U2 was removed from F_x.

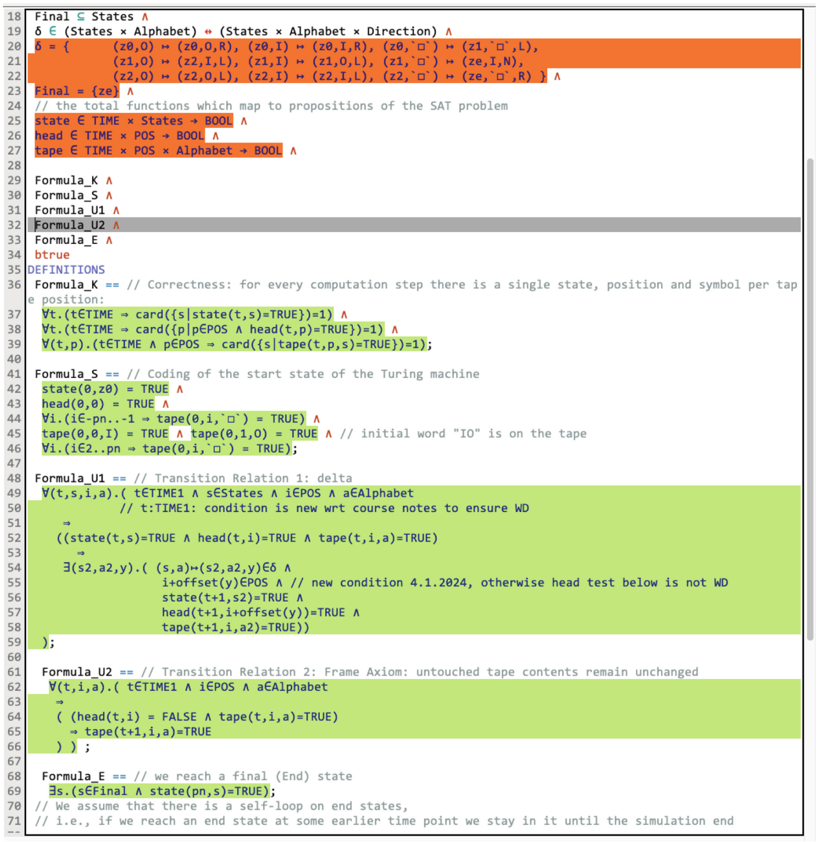
As we have seen, ProB's default solver can solve and visualise our B model in Fig. 1, and thus implicitly solve the underlying SAT problem. The underlying SAT problem in Fig. 1 has 1664 variables and a solution is found in about two seconds (when increasing solver strength preference to ensure cardinality constraints are all reified, cf. [18]). But ProB is *not* a dedicated SAT solver, and will certainly struggle for larger underlying SAT problems.

ProB also provides other constraint solving backends. In particular, ProB's Kodkod backend [35] translates B models to SAT via the Kodkod library [41]. It is an ahead-of-time static translation for a first-order subset of B, which unfortunately fails here, in particular because δ is not a binary relation. So this backend is not applicable to our encoding of Cook's theorem. Unfortunately, we were also not able to successfully apply ProB's Z3 backend [37] here. (We provide a detailed discussion of these backends in Sect. 6.)

So, given that our B model actually specifies the generation of a SAT formula, wouldn't it be nice if we somehow could generate F_x on the fly in B and then call a dedicated SAT solver on F_x?

This is exactly the contribution of this paper: a technique to process the above B model by a combination of ProB's default solver with a dedicated SAT solver. The approach enables one to use the full power of B, including higher-order functions and relations, during pre- and post-processing, while still using a propositional logic SAT solver for the core solving. Figure 2 shows parts of the B model, highlighting in green the parts that were expanded and translated for the SAT solver and in red the parts that were fully processed by ProB's solver.

[4] Implementing adding 1 to a binary number; but the output is of no relevance here.



```
Final ⊆ States ∧
δ ∈ (States × Alphabet) ⇸ (States × Alphabet × Direction) ∧
δ = {      (z0,O) ↦ (z0,O,R), (z0,I) ↦ (z0,I,R), (z0,`□`) ↦ (z1,`□`,L),
           (z1,O) ↦ (z2,I,L), (z1,I) ↦ (z1,O,L), (z1,`□`) ↦ (ze,I,N),
           (z2,O) ↦ (z2,O,L), (z2,I) ↦ (z2,I,L), (z2,`□`) ↦ (ze,`□`,R) } ∧
Final = {ze} ∧
// the total functions which map to propositions of the SAT problem
state ∈ TIME × States → BOOL ∧
head ∈ TIME × POS → BOOL ∧
tape ∈ TIME × POS × Alphabet → BOOL ∧

Formula_K ∧
Formula_S ∧
Formula_U1 ∧
Formula_U2 ∧
Formula_E ∧
btrue
DEFINITIONS
Formula_K == // Correctness: for every computation step there is a single state, position and symbol per tap
e position:
  ∀t.(t∈TIME ⇒ card({s|state(t,s)=TRUE})=1) ∧
  ∀t.(t∈TIME ⇒ card({p|p∈POS ∧ head(t,p)=TRUE})=1) ∧
  ∀(t,p).(t∈TIME ∧ p∈POS ⇒ card({s|tape(t,p,s)=TRUE})=1);

Formula_S == // Coding of the start state of the Turing machine
  state(0,z0) = TRUE ∧
  head(0,0) = TRUE ∧
  ∀i.(i∈-pn..-1 ⇒ tape(0,i,`□`) = TRUE) ∧
  tape(0,0,I) = TRUE ∧ tape(0,1,O) = TRUE ∧ // initial word "IO" is on the tape
  ∀i.(i∈2..pn ⇒ tape(0,i,`□`) = TRUE);

Formula_U1 == // Transition Relation 1: delta
  ∀(t,s,i,a).( t∈TIME1 ∧ s∈States ∧ i∈POS ∧ a∈Alphabet
           // t:TIME1: condition is new wrt course notes to ensure WD
     ⇒
   ((state(t,s)=TRUE ∧ head(t,i)=TRUE ∧ tape(t,i,a)=TRUE)
      ⇒
    ∃(s2,a2,y).( (s,a)↦(s2,a2,y)∈δ ∧
                 i+offset(y)∈POS ∧ // new condition 4.1.2024, otherwise head test below is not WD
                 state(t+1,s2)=TRUE ∧
                 head(t+1,i+offset(y))=TRUE ∧
                 tape(t+1,i,a2)=TRUE))
  );

 Formula_U2 == // Transition Relation 2: Frame Axiom: untouched tape contents remain unchanged
   ∀(t,i,a).( t∈TIME1 ∧ i∈POS ∧ a∈Alphabet
    ⇒
    ( (head(t,i) = FALSE ∧ tape(t,i,a)=TRUE)
      ⇒ tape(t+1,i,a)=TRUE
    ) ) ;

 Formula_E == // we reach a final (End) state
   ∃s.(s∈Final ∧ state(pn,s)=TRUE);
// We assume that there is a self-loop on end states,
// i.e., if we reach an end state at some earlier time point we stay in it until the simulation end
```

Fig. 2. B2SAT Translation Coverage Feedback in PROB: green parts were translated to SAT, red parts were processed by regular constraint solving

3 The B2SAT Approach

The B2SAT approach is not an ahead-of-time translation to SAT like [35] or [21], but a dynamic translation during solving with PROB's default solver.

PROB's default solver is written in Prolog and scales to very large and complex data values. It has good deterministic propagation and is ideal for animation and data validation [30]. The boolean part of PROB's solver is inspired by [19,20], but is not based on CNF and is not using watched literals. More importantly, it unfortunately has no clause learning and conflict analysis.

The essence of the B2SAT approach is to intertwine a B to CNF conversion algorithm with PROB's constraint solver. PROB's constraint solver thus runs both before and after a SAT solver is called on the CNF conversion. PROB's solver can thus be used to expand quantifiers and pre-compute complex expressions, without which a SAT conversion would be impossible. The solver can also run after a SAT solution has been found, to check additional constraints, perform additional computations (e.g., for visualising the result) or drive optimisation.

The approach is depicted in Fig. 3 and consists of the following phases:

- the deterministic propagation phase(s) of ProB's solver: it performs deterministic propagations and can expand quantifiers and total functions. It actually consists of two phases: in the first one it tries to generate only fully known values and tries to represent known sets as AVL trees [22] for efficient lookup. The second phase is still deterministic, but can also generate partially known values (like our total functions *state*, *head* and *tape*).
- a compilation phase, whereby static values are inlined. ProB's compilation is normally used for symbolic values (like infinite functions), creating a closure where all referenced values are inlined. This closure can then be evaluated later, without having access to the original state. Here we perform the compilation explicitly to simplify the formulas, in order to enable the next phase.
- the B to CNF conversion proper, which can translate a subset of B to propositional logic in conjunctive normal form. This phase currently supports: equalities and inequalities between boolean variables and constants, all logical connectives and some cardinality constraints (see below). Subformulas that cannot be solved are sent to ProB's default solver and linked to the CNF via an auxiliary propositional variable.
- solving phase, where the generated CNF is sent to an external SAT solver.
- propagation of the SAT solution to B, by progressively "grounding" the B values and predicates linked to the propositional variables.
- complete constraint solving, solving the pending constraints in B by now performing the regular non-deterministic propagations and enumerations of ProB. In case of failure, we backtrack and add additional SAT constraints to prevent the unfruitful solution.

Let us look how this works on this simple formula (cf. Fig. 3)
$f \in 1\,..\,n \rightarrow \mathit{BOOL} \wedge n = 3 \wedge f(1) = \mathit{TRUE} \wedge (\forall i.i \in 2\,..\,n \Rightarrow f(i) \neq f(i-1))$:

- in the deterministic phase the value of n is set to 3 and the value of f is partially computed to the set $\{1 \mapsto F1, 2 \mapsto F2, 3 \mapsto F3\}$, where $F1, F2, F3$ are Prolog variables. The universal quantifier is also expanded into two conjuncts $f(2) \neq f(1)$ and $f(3) \neq f(2)$.
- the remaining formula is compiled, inlining the values for f and n and precompiling the function lookups. This results in the formula $F1 = \mathit{TRUE} \wedge F2 \neq F1 \wedge F3 \neq F2$.
- the formula is translated into a CNF over the propositional variables $F1$, $F2$, $F3$ resulting in five clauses $\{F1, \neg F2 \vee \neg F1, F1 \vee F2, \neg F3 \vee \neg F2, F2 \vee F3\}$.
- this CNF is sent to a SAT solver, which computes the model $F1, \neg F2, F3$.
- this model is propagated to B, transforming the partial value of f into a full value $\{1 \mapsto \mathit{TRUE}, 2 \mapsto \mathit{FALSE}, 3 \mapsto \mathit{TRUE}\}$.
- in this case no further constraint solving is required. For example, if we had an additional conjunct $m = \mathit{card}(f \rhd \{\mathit{TRUE}\})$ this would be computed in this phase, resulting in $m = 2$.

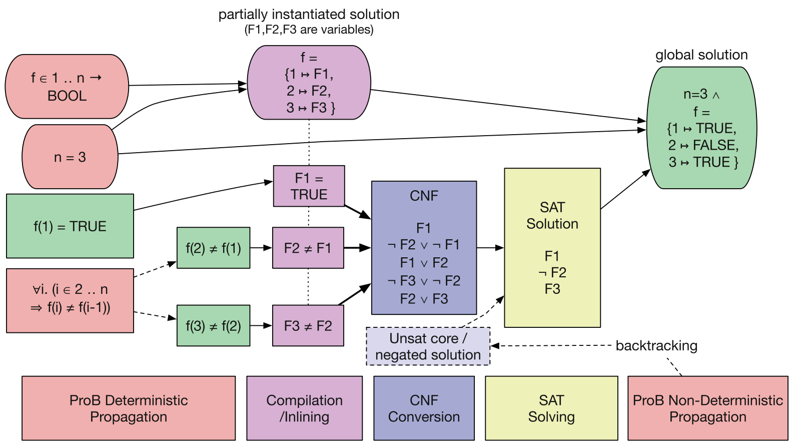

partially instantiated solution
(F1,F2,F3 are variables)
f ∈ 1 .. n → BOOL
f = {1 ↦ F1, 2 ↦ F2, 3 ↦ F3 }
global solution
n=3 ∧ f = {1 ↦ TRUE, 2 ↦ FALSE, 3 ↦ TRUE }
n = 3
F1 = TRUE
f(1) = TRUE
CNF F1 ¬ F2 ∨ ¬ F1 F1 ∨ F2 ¬ F3 ∨ ¬ F2 F2 ∨ F3
SAT Solution F1 ¬ F2 F3
f(2) ≠ f(1)
F2 ≠ F1
∀i. (i ∈ 2 .. n ⇒ f(i) ≠ f(i-1))
f(3) ≠ f(2)
F3 ≠ F2
Unsat core / negated solution
backtracking
ProB Deterministic Propagation
Compilation /Inlining
CNF Conversion
SAT Solving
ProB Non-Deterministic Propagation

Fig. 3. Solving Process Illustrated on an Example

Calling the SAT Solver. To send the CNF to the SAT solver we build on the Prolog interface to MiniSat from [7]. This interface was ported to SICStus Prolog by Sebastian Krings and adapted for recent versions of the Glucose SAT solver 4.0.[5] We are also working on targeting other SAT solvers, e.g., Kissat. Note that we call the SAT solvers directly on the generated CNF, without overhead. We have also extended our Z3 interface [37] to be able to send and solve SAT formulas in CNF (rather than SMT formulas).

Cardinality Constraints. The new B2SAT backend is of particular interest for finding solutions to complex constraints. In many of these cases one wants to minimize an objective function, often in the form of minimizing or maximizing the cardinality of a set. Also in Sect. 2 we required cardinality constraints for the subformula K in Cook's theorem.

To enable these uses of B2SAT the CNF conversion phase supports constraints of the form $card(\{x, \dots \mid P\}) \circ Expr$ where $\circ \in \{\leq, <, =, \geq, >\}$. For this conversion to work, we need to be able to extract a finite set of distinct candidates for the set $\{x, \dots \mid P\}$. This works by re-using the quantifier instantiation technique used above, expanding $\exists x.(P)$ into a disjunction, and checking that each disjunct corresponds to a unique candidate element of the set.

For example, let us examine the formula $f \in 1..3 \rightarrow \mathit{BOOL} \wedge f(1) = \mathit{TRUE} \wedge card(\{i | i \in 1 \,..\, 3 \wedge f(i) = \mathit{FALSE}\}) = 1$. As above, we would generate three propositional logic variables $F1, F2, F3$ for the contents of f. In this case quantifier expansion will create three candidate disjuncts for the set $\{i | i \in 1..3 \wedge f(i) = \mathit{FALSE}\}$: $f(1) = \mathit{FALSE}$, $f(2) = \mathit{FALSE}$ and $f(3) = \mathit{FALSE}$. This gets translated into three propositional logic literals $\neg F1, \neg F2, \neg F3$. We

[5] See https://www.labri.fr/perso/lsimon/research/glucose/.

now have to encode that exactly one of these three literals is true, e.g., as follows in CNF: $\{\neg F1 \vee \neg F2 \vee \neg F3, F1 \vee F2, F1 \vee F3, F2 \vee F3\}$.

Once we have a list of candidates of a set S as propositional logic literals $L_1, \ldots, L_k$, we need to encode the various cardinality constraints:

- $card(S) = 0$ or $card(S) < 1$ or $card(S) \leq 0$ (empty set) is $\{\neg L_1, \ldots, \neg L_k\}$.
- $card(S) = k$ or $card(S) \geq k$ (complete set) is simply $\{L_1, \ldots, L_k\}$.
- $card(S) < 0$ or $card(S) > k$ or similar unsatisfiable constraints: we generate a contradiction $\{\bot\}$.
- $card(S) \geq 1$ or similar (at least one): $\{L_1 \vee \ldots \vee L_k\}$.
- $card(S) < k$ or similar (not complete set): $\{\neg L_1 \vee \ldots \vee \neg L_k\}$.
- for the other cases we generate a sequential counter, counting the number of true literals among $L_1, \ldots, L_k$, as described in [23].

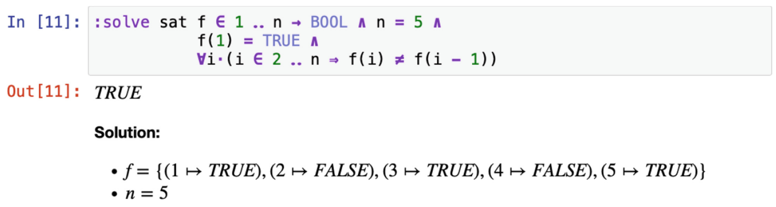
There are many works on how to encode cardinality constraints in SAT (e.g., [39,44]), but thus far we have fared well with the sequential counter encoding recommended by Knuth [23].

Tooling Extensions. We have implemented several ways in ProB to interact with the new solver backend. First, in the ProB console you have the new commands `:sat`, `:sat-z3`, `:sat-double-check` and `:sat-z3-double-check`. The first can be used to solve a predicate with Glucose, the second with Z3 [9] as SAT solver. The last two commands double-check the solution using ProB's default solver. These commands are used in ProB's integration tests.

Here is one of our examples in the console (started via `probcli --repl`):

```
:sat f:1..n --> BOOL & n=3 & f(1)=TRUE & !i.(i:2..n => f(i) /= f(i-1))
PREDICATE is TRUE
Solution:
       f = {(1|->TRUE),(2|->FALSE),(3|->TRUE)} &
       n = 3
```

It is possible to use the command `+:sat #file=FILE+` to load the predicate from a file. We have also made our solver available within ProB's Jupyter kernel [15], as the following screenshot shows:



```
In [11]: :solve sat f ∈ 1 ‥ n → BOOL ∧ n = 5 ∧
                    f(1) = TRUE ∧
                    ∀i·(i ∈ 2 ‥ n ⇒ f(i) ≠ f(i − 1))
Out[11]: TRUE

         Solution:

           • f = {(1 ↦ TRUE), (2 ↦ FALSE), (3 ↦ TRUE), (4 ↦ FALSE), (5 ↦ TRUE)}
           • n = 5
```

The backend can also be used to solve the properties (aka axioms) of B and Event-B models by setting the `SOLVER_FOR_PROPERTIES` preference. This preference can currently take the values: `prob`, `sat`, `sat-z3`, `kodkod`, `z3`, `cdclt`.[6]

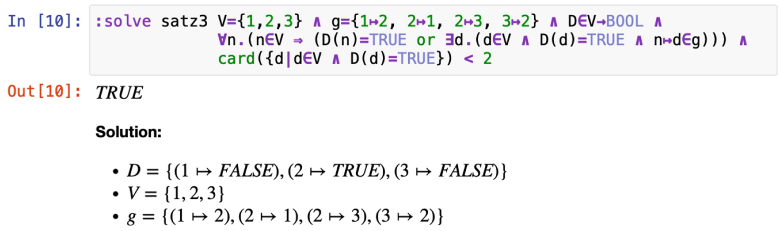
[6] The default value is `prob` while `kodkod` will use Kodkod via [35], `z3` the Z3 backend [37] and `cdclt` a Prolog implementation of SMT solving [37].

Here `sat` will use our new B2Sat backend using the default SAT solver (glucose) and `sat-z3` will use B2Sat with Z3 as SAT solver. This feature is also available for the other state-based formalisms supported by ProB, e.g., TLA$^+$ and Z.[7]

4 Applications and Experiments

Dominating Sets. Dominating sets have various practical applications. In our context, they are relevant in biological models of leaves (e.g., [40]) as well as for data generation in railway topologies. Given a graph $g \subset V \times V$ over set of nodes V, a *dominating set* is a set of nodes $D \subseteq V$ such that every node is either in D or has a neighbour in D: $\forall n.(n \in V \setminus D \Rightarrow \exists d.(d \in D \wedge n \mapsto d \in g))$.

We can encode the above formula for B2Sat. Currently, we still need to rewrite our set D as a function to BOOL (in future we will remove this restriction). To find a minimal dominating set we can add additional constraints $card(\{d \mid d \in V \wedge D(d) = TRUE\}) < b$, trying to find smaller and smaller solutions until we have found a minimal dominating set:



```
In [10]: :solve satz3 V={1,2,3} ∧ g={1↦2, 2↦1, 2↦3, 3↦2} ∧ D∈V→BOOL ∧
                      ∀n.(n∈V ⇒ (D(n)=TRUE or ∃d.(d∈V ∧ D(d)=TRUE ∧ n↦d∈g))) ∧
                      card({d|d∈V ∧ D(d)=TRUE}) < 2
Out[10]: TRUE
```

Solution:

- $D = \{(1 \mapsto FALSE), (2 \mapsto TRUE), (3 \mapsto FALSE)\}$
- $V = \{1, 2, 3\}$
- $g = \{(1 \mapsto 2), (2 \mapsto 1), (2 \mapsto 3), (3 \mapsto 2)\}$

For minimisation or optimisation one often resorts to cardinality constraints. Here the minimisation was done by "hand", but ProB also has a `MINIMIZE` predicate which can be used to automate the process.

The above constraints can be solved with the default solver of ProB. However, for bigger graphs the problem gets exponentially more complex (finding a minimal dominating set is an NP-complete problem). The left of Fig. 4 shows a minimal dominating set computed by B2Sat for a larger graph, representing a leaf, where the default solver's runtime becomes intractable. In practical applications one often needs variations or extensions of the dominating sets concept:

1. instead of considering only the direct neighbours, one can go k hops before reaching a dominating set element. D is then called a k-hop dominating set.
2. one may require additional properties of D, e.g., that it be connected. From a connected dominating set one can extract a spanning tree.

The first one is very easy to express in B: simply apply the `iterate` operator on g before applying the universal quantifier. Connectedness can also be easily expressed. The right of Fig. 4 shows a minimal 1-hop connected dominating set for the same graph. This example stems from a biological application, to study suitable vein structures of leaves.

[7] A TLA$^+$ example is available at https://prob.hhu.de/w/index.php?title=B2SAT.

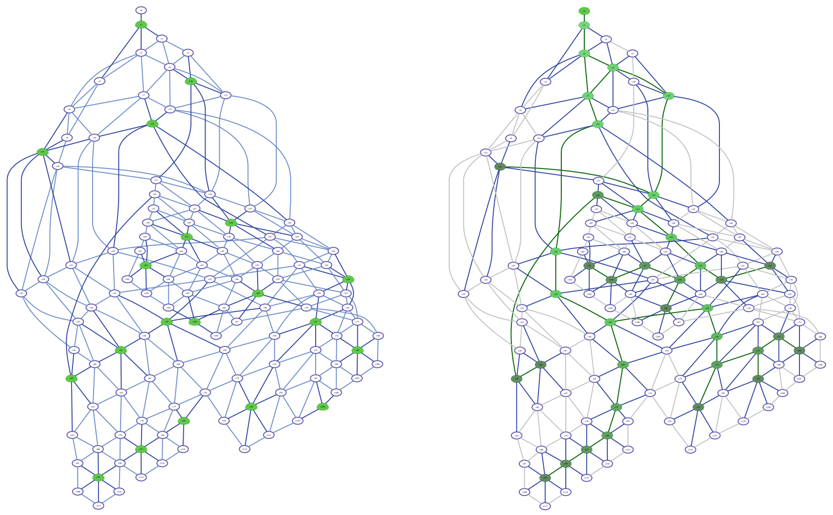


Fig. 4. Non-connected and connected 1-hop minimal dominating set. Green nodes are part of the dominating set. The graph represents a biological leaf. (Color figure online)

Similar issues can also appear in railway applications, e.g., for placing balises on a track to ensure certain safety criteria. As a proof of concept, we have solved an artificial problem on real data. We have used ProB to read in RailML data of the Oslo main station. The import uses the expressivity of B, also performing subsidiary rule validation [16]. The task was to place balises on the track which ensures that a train must encounter a balise at least every three blocks. (A very recent article [31] discusses related data generation problems for railML.)

With B2Sat we could produce a more efficient version of our time-tabling tool [38]. We hope that our backend is also applicable to other verification tasks, e.g., for interlockings. This is related to techniques like Prover iLock [3,4] or HLL [5,13]. Our hope is to make such verification available while using the full expressive power of B. We also want to address verification of B hardware models [12,42], in particular the CLEARSY safety platform [27] (LChip).

Crowded Chessboard. The crowded chessboard is a more than 100 year old problem from [10]. The purpose is to place a maximal number of chess pieces on a board, so that no piece attacks a piece of the same kind. In [26] we tried various approaches to solve the problem. In particular we developed a precursor of the present work, integrating ProB with Kodkod differently than in [35]. While better than the SMT and CLP(FD) encodings in [26], the approach was not very user-friendly (requiring explicit annotations) and considerably slower than B2Sat: the solving time for n = 8 is 19 s compared to 0.5 s with B2Sat. Note that our encoding is fully readable and is similar in performance to the direct SAT encoding from [26], while we can easily inspect, double-check and visualise the solution using ProB.

5 Experiments

Below we conducted an empirical evaluation of B2Sat.[8] Table 1 contains the run times of our new backend. It shows times for pre-processing (quantifier expansion) and conversion to CNF (second column), times for SAT solving proper (with Glucose, column 5), and total solving times (including post-processing and times from columns 2 and 5).

All benchmarks were run on a Macbook Air with M2 processor, 24 GB RAM and version 1.13.1-beta1 of ProB compiled with SICStus Prolog 4.8.0. We used Z3 in version 4.13.0.0 and Glucose in version 4.0. For the Kodkod backend we also used Glucose as SAT solver to enable a fair comparison. All times are wall times in milliseconds (ms) and the timeout was set at 20 s (but B2SAT does not yet support time-outs during the SAT solver runs).

The benchmarks contain the examples from above: bounded model checking of the Turing machine from Fig. 1 for 20 steps (TuringMachine_Cook_20), the crowded chessboard puzzle for an 8×8 chessboard (CrowdedChessBoard), dominating sets for leaves (DominatingSet_Middle), and balise placement on the Oslo main station. We also included three pure SAT problems (blocksworld and uuf) in B, to measure the overhead when writing SAT problems in B rather than in CNF format. We have also included some benchmarks from the IDP-Z3 [6] system, which we translated to B: queens, transitive closure, and pigeon hole. The translation was straightforward. These IDP models use quantifiers instead of natural B operators (e.g., `perm` for queens or `closure1` for the transitive closure), which would be considerably faster in B. Still, the models are a good way to evaluate B2Sat, whose results are very good compared to the results on the IDP-Z3 Github site.[9]

Other Backends. The comparison with other backends of ProB are in Table 2. As one can see, the Kodkod backend [35] was only applicable to 5 of the 14 examples. Some of the examples can be rewritten to make the backend applicable (see below). When applicable, however, it is often considerably slower than B2Sat, including for the three SAT problems.

The default CLP(FD) backend of ProB can solve 7 out of the 14 examples. The Z3 SMT backend can only solve 3 of the 14 examples; the treatment of quantifiers and cardinality constraints is a weak spot of this backend. Z3 can still be very useful as a SAT solver, as we explain below.

[8] The benchmarks and a Makefile to run the benchmarks are available at:https://zenodo.org/records/12180216.

[9] https://gitlab.com/krr/IDP-Z3/-/blob/main/tests/Benchmark/results.md (consulted Feb. 8th, 2024): queens 14 5.8 s, queens 24 45.8 s, pigeon_mx 100 2.8 s, transitive_closure 50 66.9 secs.

Table 1. B2SAT Backend of PROB: (1) B to CNF Pre-Processing, (2) Glucose SAT Solving and (3) Total Walltime including Post-Processing

	(1) B $\rightsquigarrow$ CNF			(2) Glucose		(3) Total
FILE	Time (ms)	Clauses	Vars	Time (ms)	Status	Time (ms)
pigeon_30	293	16590	2670	21	sat	327
transitive_closure_50	1476	15857	2450	9	sat	1501
queens_14	968	5838	196	5	sat	988
queens_24	4828	30568	576	54	sat	4906
blocksworld1	2	953	116	1	sat	176
blocksworld2	3	954	116	0	unsat	169
uuf-250-016	3	1065	250	1336	unsat	1573
DominatingSet_Middle_lt13	14	1506	790	2	sat	34
DominatingSet_Middle_lt12	15	1407	740	6	unsat	39
OSLO_no_card	50	288	288	1	sat	78
OSLO_card_lt_135	67	59824	30123	742	sat	838
OSLO_dom_edge_lt_181	36	100555	50504	16064	sat	16130
CrowdedChessBoard	401	25910	11331	63	sat	501
TuringMachine_Cook_20	6450	50333	11064	121	sat	6594

Table 2. Total Walltime for Solving with various Backends of PROB. ✠ stands for unknown, ✓ for the correct result.

	B2SAT		Kodkod		Default		Z3		CDCLT	
FILE	Status	ms	Stat	ms	Stat	ms	Stat	ms	Stat	ms
pigeon_30	sat	327	✠	21	✓	396	✠	2621	✓	9036
transitive_closure_50	sat	1501	✠	31	✓	6963	✠	20988	✓	7178
queens_14	sat	988	✠	15	✓	9073	✠	20164	✠	20095
queens_24	sat	4906	✠	15	✠	20123	✠	20361	✠	20085
blocksworld1	sat	176	✓	1114	✓	221	✓	2334	✓	387
blocksworld2	unsat	169	✓	450	✓	179	✓	2031	✓	346
uuf-250-016	unsat	1573	✓	6249	✠	20257	✓	5421	✠	20567
DominatingSet_Middle_lt13	sat	34	✓	939	✓	92	✠	21504	✓	139
DominatingSet_Middle_lt12	unsat	39	✓	732	✠	20057	✠	21419	✠	20076
OSLO_no_card	sat	78	✠	33	✠	20081	✠	21294	✠	20079
OSLO_card_lt_135	sat	838	✠	30	✠	20064	✠	21522	✠	20103
OSLO_dom_edge_lt_181	sat	16130	✠	28	✠	20116	✠	21245	✠	20106
CrowdedChessBoard	sat	501	✠	25	✠	20053	✠	20941	✠	20098
TuringMachine_Cook_20	sat	6594	✠	41	✓	8699	✠	141	✓	10928

Other SAT Solvers. To keep the tables readable, we have not included runtimes when using Z3 instead of Glucose for B2SAT in Fig 1. For most smaller examples, Glucose is much faster than Z3 (e.g., 21 ms vs. 231 ms for pigeon_30). This is probably because the Prolog SAT interface based on [7] is faster than Z3's C++ interface. For complex examples, however, Z3 can be a useful alternative SAT solver. For example, for OSLO_dom_edge_lt_181 it is four times faster than Glucose. It is good to have a variety of SAT solvers at our disposal; especially for optimisation, where solving time increases when we approach the optimum.

In summary, the tables show that for the benchmarks above, B2SAT is a considerable improvement over existing backends, and opens up new application areas for B. There is still a performance bottleneck in the compilation phase, as can be seen in the queens, transitive closure and Turing examples in Table 1. The overhead is due to partially instantiated data values having linear rather than logarithmic access in PROB. We hope to reduce this overhead considerably in the future, e.g., by also using AVL trees for partially instantiated values.

Tables 1 and 2 are biased: we only study examples which can be solved by B2SAT. Also, some benchmarks can be rewritten for Kodkod by replacing strings with enumerated sets, rewriting functions to predicates, or adding additional constraints to make the bounds finite or remove higher-order constructs by hand. For example, by rewriting the pigeon_30 example and removing the higher-order function it can be solved in 596 ms. The purpose of the experiments is to show that there are applications where B2SAT is very effective; it is *not* to study the performance for a representative set of benchmark programs.

6 Related and Future Work

Other B Backends. We can compare B2SAT with the backends from Table 2:

- PROB's default solver is based on constraint logic programming. As mentioned, it scales to very large and complex data values and has been used in industry for B specifications with up to 9 million lines of B. It has good deterministic propagation, can deal symbolically with infinite values and is well suited for animation and data validation. The boolean solver was inspired by [19,20], but without watched literals. Also, there is no clause learning nor conflict analysis.
- The CDCL(T) backend [37] is a Prolog SMT-style solver built on top of PROB's default solver. It does have clause learning and conflict analysis, but its performance as a SAT solver is far from state-of-the-art SAT solvers. It is useful for symbolic verification tasks, but as Table 2 shows not for the constraint solving and optimisation tasks here.
- The Z3 SMT backend [37] is based on a translation of B to SMT-LIB. It works better with unsatisfiable formulas than for model finding of satisfiable ones. The backend is good for symbolic verification tasks, but has still considerable restrictions (cardinality, quantifiers, finite B values often get translated to infinite ones in SMT-LIB, ...). As such it is not suited as an animation engine and as Table 2 shows not for the benchmarks here.

The Apalache symbolic model checker [25] for TLA$^+$ uses Z3 [9] as backend, but with an encoding tailored for finite sets. As such it is better suited for model finding. Indeed, we were able to solve a small dominating set example in TLA$^+$ but not DominatingSet_Middle_lt13 from Table 1.[10]
- The Kodkod backend [35] uses the Kodkod library [41] to perform an indirect translation of B to SAT (via the relational logic API of [41]). When applicable, it can be very effective and much more efficient than ProB's default solver. It has, however, limited applicability (no sets of sets, no higher-order relations or functions, restrictions to binary relations).

The Kodkod backend [35] is the closest to our approach, and we want to clarify the important differences:

- [35] is a static ahead-of-time translation on the AST (abstract syntax tree). As such there is no expansion prior to translation, meaning we cannot use many of B's nice features (higher-order, ...) to set-up the constraints. B2Sat is dynamic (just-in-time, e.g., after quantifier expansions) and can process a mixture of AST and partially instantiated values.
- [35] can translate integers and more operators to SAT than B2Sat.
- There is an overhead in the generated SAT problem with [35] (see Table 2 for pure SAT problems).
- there is an issue with integer overflows in Kodkod, which is not easy to solve (meaning the current backend [35] is not sound for cardinality constraints or some integer membership constraints).
- [35] cannot deal with higher-order relations, nor with ternary relations (see pigeon_30 in Table 2).

Other Languages Translating to SAT. The Alloy analyzer [21] uses the Kodkod library, and is again a static ahead-of-time translation to SAT. Arby [33] is an embedding of Alloy into Ruby. One could thus write an Arby program to expand the Turing machine from Sect. 2 into an Alloy model, which in turn would get translated to SAT. Our approach is to use logic, mathematics and the B language to set up SAT constraints (rather than a separate scripting language).

Answer Set Programming (ASP) [11] starts off from a logic program, which usually (see [14]) gets transformed via a grounding phase to a SAT problem. ASP builds on a non-monotonic semantics, while our approach is rooted in mathematical logic with classical monotonic negation and with access to theorem provers.

The Picat [45] logic-based language one can use SAT solvers for constraint solving. A related approach is IDP-Z3 [32], based on inductive definitions rather than logic programs. IDP-Z3 is a re-implementation of [43]. We have used some IDP-Z3 benchmarks above. SMT-LIB itself, in particular when using eager solving, is also related to B2Sat. Apart from expressivity, a major difference is that in B2Sat a separate constraint-based solver is driving the translation to SAT. This increases the specifications that can be handled and the pre-processing that can take place (see also [17]). Indeed, in the crowded chessboard example the direct SMT-LIB solutions were not effective [26] (in contrast to B2Sat).

[10] Apalache version 0.44.10 produced a StackOverflow error after 122 s.

Future Work. We wish to extend the subset of B which can be translated to SAT (integer operators, finite-domain variables, sets,...). We also want to keep track when propagation of a SAT solution fails in ProB, to then compute the unsat core to add it as a learned clause. We plan to target other SAT solvers, like Kissat. and would like to target SMT rather than SAT. As we saw in Table 2, the current Z3 backend does not work well for model finding or optimisation. By generating SMT-LIB without quantifiers this could be much improved.

In the future we wish to make the optimisation process more efficient. ProB already has the functions MAXIMIZE and MINIMIZE, but we wish to use incremental SAT solving and other algorithms from the SAT community [24].

In summary, we have presented a new bare metal SAT backend for B. We have shown how it can be applied almost out of the box to a mathematical rendering of Cook's theorem. With our new backend one can use the full power of B to pre- and post-process higher-order data and properties, solve and optimise complex problems and use the B tooling infrastructure to visualise solutions and double check solutions with other backends. We hope that this leads to readable, maintainable and efficient SAT applications with state-based formal methods like B, Z or TLA+. While this approach is certainly not a universal technique, it enables a wide variety of new applications: graph matching for machine learning, dominating sets for biological applications, hardware modelling and verification, data generation for industrial railway applications, bounded model checking for railway interlockings, and many more.

Acknowledgements. Thanks to Sebastian Krings for the initial port of the miniSat-Prolog interface [7] to SICStus. I also thank Joshua Schmidt for enabling Z3 as SAT solver and providing useful feedback on the article. Finally, thanks to Jan Roßbach who is working to make more SAT solvers available for ProB.

Data Availability Statement. The models and instructions on how to run the benchmarks are available at https://stups.hhu-hosting.de/models/B2SAT/ and in a Zenodo archive with DOI `10.5281/zenodo.12180216`. The latest version is available at https://zenodo.org/doi/10.5281/zenodo.12166662.

References

1. Abrial, J.-R.: The B-Book. Cambridge University Press (1996)
2. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Borälv, A.: The industrial success of verification tools based on stålmarck's method. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 7–10. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_3
4. Borälv, A.: Case study: formal verification of a computerized railway interlocking. Formal Aspects Comput. **10**(4), 338–360 (1998)
5. Breton, N., Fonteneau, Y.: S3: proving the safety of critical systems. In Proceedings RSSRail **2016**, 231–242 (2016)

6. Carbonnelle, P., Vandevelde, S., Vennekens, J., Denecker, M.: IDP-Z3: a reasoning engine for FO(.). CoRR, abs/2202.00343 (2022)
7. Codish, M., Lagoon, V., Stuckey, P.J.: Logic programming with satisfiability. Theory Pract. Logic Program. **8**(1), 121–128 (2008)
8. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC 1971, pp. 151–158, New York, NY, USA (1971). Association for Computing Machinery
9. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
10. Dudeney, H.E.: Amusements in Mathematics (1917). https://www.gutenberg.org/ebooks/16713
11. Erdem, E., Gelfond, M., Leone, N.: Applications of answer set programming. AI Mag. **37**(3), 53–68 (2016)
12. Evans, N., Ifill, W.: Hardware verification and beyond: using B at AWE. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 260–261. Springer, Heidelberg (2006). https://doi.org/10.1007/11955757_24
13. Ge, N., Jenn, E., Breton, N., Fonteneau, Y.: Integrated formal verification of safety-critical software. Int. J. Softw. Tools Technol. Transf. **20**(4), 423–440 (2018)
14. Gebser, M., Leone, N., Maratea, M., Perri, S., Ricca, F., Schaub, T.: Evaluation techniques and systems for answer set programming: a survey. In: Lang, J. (ed.) Proceedings IJCAI 2018, pp. 5450–5456 (2018). https://www.ijcai.org/
15. Geleßus, D., Leuschel, M.: ProB and Jupyter for logic, set theory, theoretical computer science and formal methods. In: Raschke, A., Méry, D., Houdek, F. (eds.) ABZ 2020. LNCS, vol. 12071, pp. 248–254. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_19
16. Gruteser, J., Leuschel, M.: Validation of railML using ProB. In: Proceedings ICECCS 2024, LNCS (June 2024). https://doi.org/10.1007/978-3-031-66456-4_13
17. Hadarean, L., Bansal, K., Jovanović, D., Barrett, C., Tinelli, C.: A tale of two solvers: eager and lazy approaches to bit-vectors. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 680–695. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_45
18. Hallerstede, S., Leuschel, M.: Constraint-based deadlock checking of high-level specifications. Theory Pract. Log. Program. **11**(4–5), 767–782 (2011)
19. Howe, J.M., King, A.: A pearl on SAT solving in prolog. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 165–174. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12251-4_13
20. Howe, J.M., King, A.: A pearl on SAT and SMT solving in Prolog. Theor. Comput. Sci. **435**, 43–55 (2012)
21. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**, 256–290 (2002)
22. Knuth, D.: The Art of Computer Programming, Volume 3. Addison-Wesley (1983)
23. Knuth, D.: The Art of Computer Programming, Volume 4, Fascicle 6: Satsfiability. Addison-Wesley (2015)
24. Kochemazov, S., Ignatiev, A., Marques-Silva, J.: Assessing progress in SAT solvers through the lens of incremental SAT. In: Li, C.-M., Manyà, F. (eds.) SAT 2021. LNCS, vol. 12831, pp. 280–298. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-80223-3_20
25. Konnov, I., Kukovec, J., Tran,T.: TLA+ model checking made symbolic. Proc. ACM Program. Lang., **3**(OOPSLA), 123:1–123:30 (2019)

26. Krings, S., Leuschel, M., Körner, P., Hallerstede, S., Hasanagić, M.: Three Is a crowd: SAT, SMT and CLP on a chessboard. In: Calimeri, F., Hamlen, K., Leone, N. (eds.) PADL 2018. LNCS, vol. 10702, pp. 63–79. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73305-0_5
27. Lecomte, T., Déharbe, D., Fournier, P., Oliveira, M.: The CLEARSY safety platform: 5 years of research, development and deployment. Sci. Comput. Program. **199**, 102524 (2020)
28. Leuschel, M.: ProB: Harnessing the power of Prolog to bring formal models and mathematics to life. Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M.V., Kowalski, R., Rossi, F. (eds.) Prolog: The Next 50 Years, LNCS 13900, vol. 13900, pp. 239–247. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-35254-6_19
29. Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805. Springer, Heidelberg (2003). https://doi.org/10.1007/b13229
30. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated property verification for large scale B models with ProB. Formal Asp. Comput. **23**(6), 683–709 (2011). https://doi.org/10.1007/s00165-010-0172-1
31. Menéndez, M.N., Germino, S., Díaz-Charris, L.D., Lutenberg, A.: Automatic railway signaling generation for railways systems described on railway markup language (railML). IEEE Trans. Intell. Transp. Syst. **25**(3), 2331–2341 (2024)
32. Mikhailov, L., Butler, M.: An approach to combining B and alloy. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 140–161. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45648-1_8
33. Milicevic, A., Efrati, I., Jackson, D.: αrby - an embedding of alloy in ruby. In: Ait Ameur, Y., Schewe, KD. (eds.) Proceedings ABZ, vol. 8477, pp. 56–71 (2014). https://doi.org/10.1007/978-3-662-43652-3_5
34. Papadimitriou, C.H.: Cook's NP-completeness paper and the dawn of the new theory. In: Kapron, B.M. (ed.) Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook, ACM Books, vol. 43, pp. 73–82. ACM (2023)
35. Plagge, D., Leuschel, M.: Validating B,Z and TLA$^+$ using ProB and kodkod. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 372–386. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_31
36. Rothe, J.: Theoretische informatik. Technical report, University of Düsseldorf (2000–2024)
37. Schmidt, J., Leuschel, M.: SMT solving for the validation of B and event-b models. Int. J. Softw. Tools Technol. Transf. **24**(6), 1043–1077 (2022)
38. Schneider, D., Leuschel, M., Witt, T.: Model-based problem solving for university timetable validation and improvement. Formal Aspects Comput. **30**(5), 545–569 (2018)
39. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005). https://doi.org/10.1007/11564751_73
40. Surlemont, M.: Solving connected dominating set variants using integer linear programming. Bachelor's thesis, Institut für Informatik, Universität Düsseldorf (2020)
41. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49
42. Voros, N.S., Snook, C.F., Hallerstede, S., Masselos, K.: Embedded system design using formal model refinement: an approach based on the combined use of UML and the B language. Design Autom. for Emb. Sys. **9**(2), 67–99 (2004)
43. Wittocx, J., Mariën, M., Denecker, M.: Grounding FO and FO(ID) with bounds. J. Artif. Intell. Res. (JAIR) **38**, 223–269 (2010)

44. Wynn, E.: A comparison of encodings for cardinality constraints in a SAT solver. CoRR, abs/1810.12975 (2018)
45. Zhou, N.: Modeling and solving graph synthesis problems using sat-encoded reachability constraints in picat. In: Formisano, A., et al. (eds.) Proceedings ICLP 2021, EPTCS, vol. 345, pp. 165–178 (2021)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

PyBDR: Set-Boundary Based Reachability Analysis Toolkit in Python

Jianqiang Ding[1,2](✉), Taoran Wu[2,3], Zhen Liang[4], and Bai Xue[2,3]

[1] Aalto University, Espoo, Finland
jianqiang.ding@aalto.fi
[2] Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
dingjianqiang0x@gmail.com, {wutr,xuebai}@ios.ac.cn
[3] University of Chinese Academy of Sciences, Beijing, China
[4] National University of Defense Technology, Hunan, China
liangzhen@nudt.edu.cn

Abstract. We present PyBDR, a Python reachability analysis toolkit based on set-boundary analysis, which centralizes on widely-adopted set propagation techniques for formal verification, controller synthesis, state estimation, etc. It employs boundary analysis of initial sets to mitigate the wrapping effect during computations, thus improving the performance of reachability analysis algorithms without significantly increasing computational costs. Beyond offering various set representations such as polytopes and zonotopes, our toolkit particularly excels in interval arithmetic by extending operations to the tensor level, enabling efficient parallel interval arithmetic computation and unifying vector and matrix intervals into a single framework. Furthermore, it features symbolic computation of derivatives of arbitrary order and evaluates them as real or interval-valued functions, which is essential for approximating behaviours of nonlinear systems at specific time instants. Its modular architecture design offers a series of building blocks that facilitate the prototype development of reachability analysis algorithms. Comparative studies showcase its strengths in handling verification tasks with large initial sets or long time horizons. The toolkit is available at https://github.com/ASAG-ISCAS/PyBDR.

1 Introduction

Reachability analysis, which mainly involves the computation of reachable sets, is an essential tool for rigorously determining the behavior of dynamical systems across different scenarios. It serves as the foundation for applications such as formal verification [6,23,36], controller synthesis [29,34], and state estimation [1]. While the precise reachable set can be characterized using sublevel sets of solutions to Hamilton-Jacobi (HJ) equations [13,27], the necessity of discretizing state

J. Ding and T. Wu—These authors contribute equally to this work.

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 140–157, 2025.
https://doi.org/10.1007/978-3-031-71177-0_10

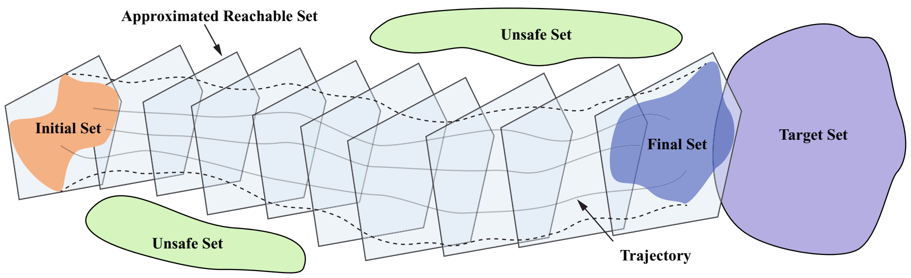


Fig. 1. Reachability analysis based on set propagation techniques.

space for numerical solving, limit their applicability to high-dimensional dynamic systems due to the escalating computational expenses linked to dimensionality. These limitations have led the control community to prefer using approximate strategies for reachability analysis, such as set propagation techniques [4].

The set propagation method, depicted in Fig. 1, extends the numerical solution of ordinary differential equations (ODEs) by using sets to represent solutions rather than precise numerical values. This method commences from an initial state set and iteratively computes sets to encompass all possible system states, thus supports the verification of specific properties like safety [3,9,12,16,17,22,30]. To expedite set operations, the method employs representations such as intervals, polytopes, and zonotopes to over-approximate the exact reachable set. However, the cumulative error from successive iterations, known as the wrapping effect [28], can lead to overly conservative state estimations, particularly for large initial sets or large time periods, potentially causing verification failures. While partitioning the initial set or adjusting the step size can mitigate wrapping effect errors, this simple strategy often incurs substantial computational expenses, rendering it impractical for refining the conservative estimates of existing reachability analysis algorithms. On the other hand, the shared algorithmic structure of set-propagation methods often allows further advancements to be built upon improving specific steps rather than overhauling the entire design. However, implementing such customized algorithms often deviate from the primary objective of existing reachability analysis tools, which prioritize user-friendly interfaces over the creation of developer-centric platforms conducive to innovative algorithmic research and development.

In this work, we introduce PyBDR, our prototype toolkit for set-boundary-based reachability analysis, developed in Python. PyBDR includes advanced set-boundary propagation methods designed to enhance reachability analysis capabilities, particularly for large initial sets and long time horizons. Based on the homeomorphism property of the solution mapping for ODEs satisfying Lipschitz conditions, the set-boundary propagation method propagates only the boundary of the initial set rather than the entire initial set itself to conduct reachability analysis [37,38]. Because the measure (or, volume) of the boundary is much smaller than the one of the entire initial set, the set-boundary propagation method will induce a smaller wrapping effect efficiently. Furthermore, to support algorithm development, we envision a paradigm where developers are empowered with a suite of accessible, modular, and versatile building blocks, such as the design of Interval Tensors. These crafted blocks aim to facilitate and streamline the iterative refinement of innovative reachability analysis algorithms. As illustrated in Fig. 2, the architecture of PyBDR features the following three core modules:

- **geometric module**: The geometric module enriches the toolkit by incorporating established conventional set representations such as intervals, polytopes, and zonotopes. It innovatively advances interval arithmetic to the tensor level with the aid of a broadcasting mechanism. This advancement enables the parallelization of operations and provides a unified framework for manipulating vector intervals, matrix intervals, and interval matrices.
- **dynamic module**: In addition to supporting linear systems, the dynamic module is specifically designed to manage nonlinear systems. It facilitates arbitrary-order derivative evaluation through symbolic computation, thereby enabling the approximation of nonlinear systems using Taylor series expansions to arbitrary degrees.
- **utility module**: To assist in the implementation of reachability analysis algorithms, we have encapsulated interfaces for commonly used optimization methods and included a visualizer module for displaying computational results.

In addition to a modular architectural design, we also conducted a comprehensive evaluation of potential programming languages aligned with our objectives. Matlab, despite its prowess in matrix and symbolic computations, was dismissed due to its reliance on commercial licensing conflicting with our commitment to open-source principles. Similarly, while C/C++ offer high performance exemplified by tools like HyPro [33] and Flow* [16], their limited flexibility in supporting academic research prototypes made them less suitable for our needs. Although Julia shows promise in scientific computation, its relatively nascent community and ecosystem compared to Python persuaded us to explore other options. Ultimately, Python emerged as our choice not only for its user-friendly syntax and support for rapid prototyping but also for its extensive community and interoperability, crucial for integrating third-party resources in the development of reachability analysis algorithms.

Related Work. Recent developments in reachability analysis have led to a range of tools emphasizing different strengths. C/C++-based tools such as SpaceEx [17] and Flow* [16] excel in efficient algorithms for both linear and/or nonlinear hybrid systems. SpaceEx integrates diverse algorithms for linear systems, while HyPro [33] focuses on convex set representation similar to LazySets. Flow* distinguishes itself with Taylor model approximation for nonlinear dynamics. However, these tools often require compilation, which can slow down rapid prototyping cycles.

In contrast, tools like CORA [3] do not require pre-run compilation, offering a wide range of algorithms for linear and nonlinear systems, including methods based on zonotopes and interval arithmetic. Attempts to port CORA's capabilities to C++ have resulted in tools like CORA/SX and SymReach, which demonstrate significant speed improvements in specific scenarios.

Python has also gained popularity in reachability analysis tools. HyLAA [8] provides discrete-time reachability algorithms for linear hybrid systems, while CommonRoad-Reach [24] combines a Python interface with a C++ core to compute reachable sets and driving corridors for autonomous vehicles in dynamic traffic, suitable for real-time applications.

Julia, known for its prowess in scientific computing, is exemplified by tools like JuliaReach [11], which provides efficient algorithms for sophisticated, high-dimensional problems. Despite Julia's performance comparable to compiled languages, its ecosystem is still developing and not as extensive as Python's.

The shift towards JIT-compiled or interpreted languages such as CORA, JuliaReach, and HyLAA reflects their flexibility in prototyping, crucial for the iterative development of algorithms. This trend underscores the community's preference for platforms that balance ease of use with computational efficiency.

The remainder of this paper is structured as follows. We in Sect. 2 detail the architecture and features of PyBDR. In Sect. 3, we illustrate the performance of our tool PyBDR. Finally, we conclude this work in Sect. 4.

2 Architecture and Features

2.1 Architecture

In this section, we present an integrated framework of our prototype tool designed to enhance the computational processes involved in reachability analysis. The framework revolves around three core modules: the geometric module, the dynamic module, and the utility module, as illustrated in Fig. 2. By leveraging the functionality of these three modules, we have integrated the implementation of several reachability analysis algorithms [2,7,37,38]. These implementations not only facilitate code reuse for the development of advanced methods but also showcase the tool's potential in supporting the creation of innovative algorithms.

Geometric Module. The geometric module of PyBDR offers various set representations, including intervals, polytopes, and zonotopes, aiming to strike a balance between computational efficiency and the precision of reachable set computations. This module provides essential operations for arithmetic operations among sets, such as Minkowski addition [21] and linear transformations. Moreover, the module supports geometric operations for converting between different set representations and computing their enclosures. A significant feature highlighted in Fig. 3 is the boundary extraction interface. This interface enables the over-approximation of the boundary of an entire set using a collection of smaller

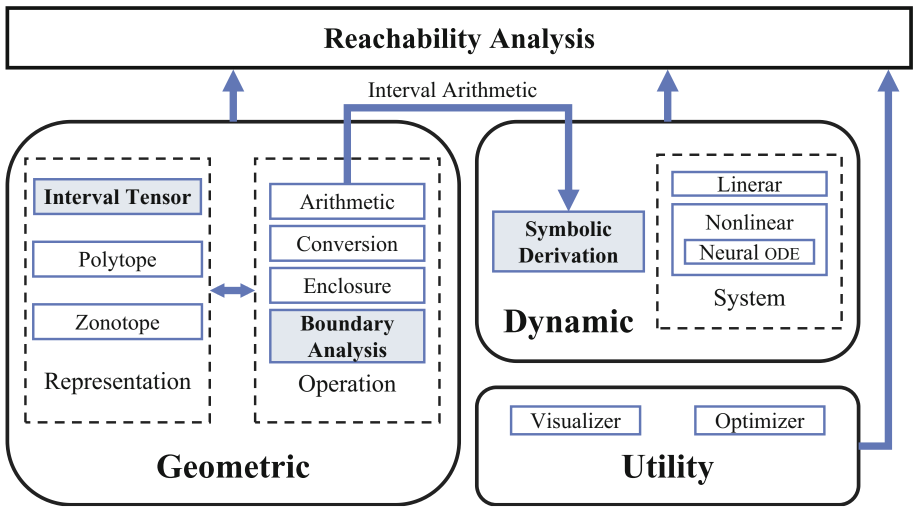


Fig. 2. Hierarchical module design in PyBDR. Solid arrows indicate functional dependencies and essential modules are highlighted with a light blue fill. (Color figure online)

geometric entities, thereby facilitating set-boundary propagation based reachability analysis. To our knowledge, PyBDR is the first reachability analysis toolkit to offer interfaces for boundary over-approximation of convex sets like zonotopes, intervals, and polytopes.

Dynamic Module. The dynamic module of PyBDR supports the definition of various types of systems, including continuous time-invariant linear systems, continuous nonlinear systems, and network-structured nonlinear systems. A notable capability of this module is its ability to analyze the behavior of Neural Ordinary Differential Equations (Neural ODEs) [14]. These systems adhere to homeomorphic mappings and can incorporate control inputs, expanding the scope of traditional reachability analysis methods.

Utility Module. The design of the utility module in PyBDR aims to offer interfaces for convex optimization problems tailored to diverse algorithmic requirements. Additionally, this module provides visualization functionalities that allow for graphical display of computed reachable sets. These visualizations enable users to intuitively analyze and evaluate the performance of the algorithm.

To provide a comprehensive overview of the advancements introduced by our tool PyBDR in reachability analysis, we present a comparative summary in Table 1. This table outlines the key characteristics of state-of-the-art reachability analysis tools alongside those of PyBDR, emphasizing the unique features and capabilities of our toolkit.

Table 1. Comparison of reachability analysis tools

Tool	Supported Systems	Principal Set Representation	Language	Additional Features	License	Latest Release
PyBDR	Linear ODEs Nonlinear ODEs Neural ODEs	Interval, Polytopes, Zonotopes	Python	Boundary Analysis, Interval Tensor, Symbolic differentiation	GPLv3	2024/04/14
CORA	Linear ODEs Nonlinear ODEs Hybrid Systems Neural Netoworks	Intervals, Polytopes, Zonotopes, Taylor Models, Polynomial Zonotopes	MATLAB	Conversion Interfaces with Other Tools	GPLv3	2024/07/01
JuliaReach	Linear ODEs Nonliear ODEs Hybrid Systems	Zonotopes, Polyhedra, Taylor Models	Julia	Lazy Sets	MIT	2023/08/30
HyLAA	Hybrid Systems with Linear ODEs	Generalized Star Representation	Python	Simulation Equivalent	GPLv3	2019/08/01
HyPro	Nonlinear Hybrid Systems	Box, Polytope, Zonotope	C++	Inexact and Exact Computation	MIT	2023/09/06
Flow*	Nonlinear Hybrid Systems	Taylor Model	C++	Adaptive Technique	GPLv3	2017/03/09

2.2 Features

Boundary Analysis. ODEs satisfying Lipschitz conditions ensure the uniqueness of evolutionary trajectories from initial states. This property, illustrated in Fig. 3, guarantees a boundary correspondence between the initial set and its reachable set throughout the system's evolution [37–39]. That is, the set reachable from the initial set's boundary is equal to the boundary of the initial set's reachable set. Therefore, the boundary of the reachable set is determined by the boundary of the initial set. A significant feature of our tool is its capability to enhance existing reachability analysis methods by focusing on the boundary analysis of the initial set. To support this capability, we have developed boundary extraction features for various common set representations.

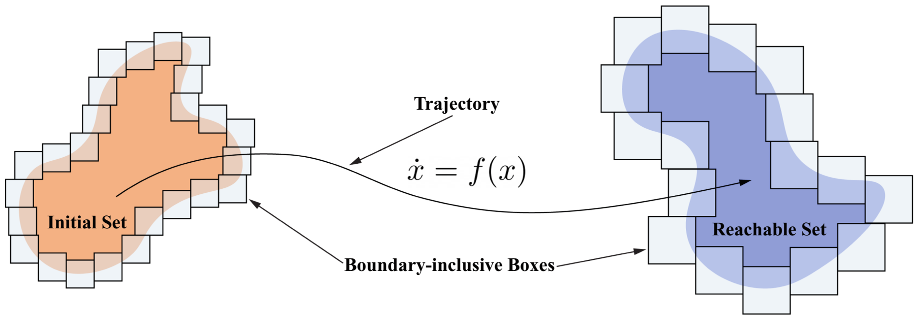


Fig. 3. Illustration of reachability analysis utilizing boundary analysis.

We offer two methods for boundary extraction, one of which utilizes the intrinsic boundary solving algorithms internally to handle the extraction of intrinsic boundaries for intervals and zonotopes [31], such as extracting 4 edges of a rectangle characterizing a two-dimensional interval.

Additionally, we incorporate the method in Realpaver [18] for boundary extraction. This method can construct a series of smaller boxes to closely enclose the exact boundary of the initial set, as depicted in Fig. 3. By strategically reducing the size of these boxes, we aim to minimize errors introduced by the wrapping effect. This meticulous selection of smaller boxes allows for a higher precision characterization of the reachable set's boundary, thereby reducing discrepancies between the computed reachable set and the actual evolution of the system. Figure 4c demonstrates that computing the reachable set using these smaller entities provides a more precise boundary approximation compared to results obtained from analyzing the entire initial set directly. This approach offers a more accurate solution for verification problems.

Listing 1. Third order Lagrange remainder calculation in PyBDR

```python
# calculate the Lagrange remainder term of the thrid order in PyBDR
xx = Interval.sum((ihx @ tx @ ihx) * ihx, axis=1)
uu = Interval.sum((ihu @ tu @ ihu) * ihu, axis=1)
err_lagr = (xx + uu) / 6
```

Listing 2. Third order Lagrange remainder calculation in CORA

```matlab
% calculate the Lagrange remainder term of third order in ↩
    CORA
error_thirdOrder_dyn = interval(zeros(obj.dim,1),zeros(obj.↩
    dim,1));
for i=1:length(ind)
```

```
    error_sum = interval(0,0);
    for j=1:length(ind{i})
        error_sum = error_sum + (dz.'*T{i,ind{i}(j)}*dz) * dz↩
            (ind{i}(j));
    end
    error_thirdOrder_dyn(i,1) = 1/6*error_sum;
end
```

Interval Tensors. Interval arithmetic is an important tool in many reachability analysis algorithms to incorporate considerations for errors during calculations. Traditionally applied to intervals, it has been extended to handle more complex data structures such as interval matrices, which represent linear systems with parametric uncertainties. This extension requires the ability to perform interval arithmetic operations in a broader context when computing reachable sets. To address this need, we have developed the Interval Tensor data structure in PyBDR. Built upon NumPy's broadcasting mechanism [19], Interval Tensor provides a versatile representation that seamlessly integrates various interval computations within a unified framework. This includes operations on interval vectors, vector intervals, interval matrices, and matrix intervals.

The Interval Tensor in PyBDR is designed to optimize computational efficiency by leveraging vectorized operations that are executed at a lower level in C, thereby minimizing the use of Python's for loops. This approach significantly enhances computational efficiency. Moreover, Interval Tensor relaxes strict shape requirements on data during computations, allowing for operations like simultaneous interval matrix multiplication with scalar matrices, as demonstrated in Listing 3. By harnessing NumPy's broadcasting mechanism, Interval Tensor improves ease of programming and enhances code readability. Compared to traditional approaches that rely heavily on explicit loops, PyBDR's implementation, as illustrated in Listings 1 and 2, demonstrates efficient computation of complex tasks such as computing the Lagrange remainder term of the third order [2]. This showcases the practical advantages of Interval Tensor in managing intricate calculations while bolstering code readability and maintainability.

In summary, Interval Tensor not only optimizes computational efficiency through vectorization but also enhances the clarity and maintainability of algorithms in PyBDR.

Listing 3. Interval tensor matrix multiplication in PyBDR

```python
# interval matrix multiplication simultaneously in PyBDR
a = Interval.rand(100, 2, 5, 4)
b = np.random.rand(4, 9)
c = a @ b
print(c.shape) # (100, 2, 5, 9)
```

In addition to enhancing code writing and readability, we compare the performance of PyBDR and CORA in different computational tasks to examine the average time consumption and accuracy of Interval Tensor in performing interval

Table 2. Comparative evaluation of PyBDR and CORA for interval arithmetic operations.

Operator	Functionality	ϵ	Avg. Time [s]		Input Intervals			
			CORA	PyBDR	$\underline{I}$	$\overline{I}$	$\underline{I}_\delta$	$\overline{I}_\delta$
$+$	addition	0	$1.01e^{-8}$	$4.68e^{-9}$	-100	100	0	100
$-$	subtraction	0	$4.02e^{-8}$	$7.28e^{-9}$	-100	100	0	100
$*$	multiplication	0	$1.96e^{-5}$	$1.54e^{-8}$	-100	100	0	100
/	division	0	$3.22e^{-5}$	$4.60e^{-8}$	-100	100	0	100
$**$	power	$1.08e^{-15}$	$1.74e^{-5}$	$2.93e^{-8}$	-100	100	0	100
$\lvert\ \rvert$	absolute	0	$1.30e^{-8}$	$2.30e^{-8}$	-100	100	0	100
@	left matrix multiplication	0	$5.74e^{-5}$	$1.15e^{-6}$	-100	100	0	100
	right matrix multiplication	0	$6.05e^{-5}$	$1.16e^{-6}$	-100	100	0	100
exp	exponential	$2.15e^{-16}$	$2.08e^{-8}$	$1.07e^{-8}$	-100	100	0	100
log	logarithm	$1.99e^{-14}$	$4.36e^{-8}$	$8.41e^{-9}$	0	100	0	100
sqrt	square root	0	$2.87e^{-8}$	$4.17e^{-9}$	0	100	0	100
sin	sine	$1.66e^{-14}$	$3.69e^{-7}$	$4.47e^{-8}$	-100	100	0	100
cos	cosine	$6.21e^{-15}$	$4.26e^{-8}$	$3.85e^{-8}$	-100	100	0	100
tan	tangent	$1.25e^{-14}$	$4.03e^{-8}$	$1.90e^{-8}$	$-\frac{\pi}{2}+0.01$	0	0	$\frac{\pi}{2}-0.01$
cot	cotangent	N/A	N/A	$4.02e^{-8}$	0.01	$\frac{\pi}{2}$	0	$\frac{\pi}{2}-0.01$
arcsin	inverse sine	$9.78e^{-16}$	$3.70e^{-8}$	$2.07e^{-8}$	-1	0	0	1
arccos	inverse cosine	$8.13e^{-15}$	$3.89e^{-8}$	$1.73e^{-8}$	-1	0	0	1
arctan	inverse tangent	$3.19e^{-14}$	$1.61e^{-8}$	$1.04e^{-8}$	-100	100	0	100
sinh	hyperbolic sine	$2.19e^{-16}$	$3.72e^{-9}$	$1.73e^{-8}$	-100	100	0	100
cosh	hyperbolic cosine	$2.20e^{-16}$	$5.43e^{-8}$	$5.18e^{-8}$	-100	100	0	100
tanh	hyperbolic tangent	$9.58e^{-15}$	$1.09e^{-8}$	$9.19e^{-9}$	-1	1	0	1
arcsinh	inverse hyperbolic sine	$9.29e^{-15}$	$1.87e^{-8}$	$2.51e^{-8}$	-100	100	0	100
arccosh	inverse hyperbolic cosine	$1.60e^{-14}$	$2.72e^{-8}$	$2.32e^{-8}$	1	10	0	10
arctanh	inverse hyperbolic tangent	$2.82e^{-15}$	$4.02e^{-8}$	$2.39e^{-8}$	-1	0	0	1

Note: N/A – not available due to the absence of *cot* implementation in CORA; ϵ – see (1);

arithmetic operations. CORA was specifically chosen as a baseline due to its use of MATLAB, an interpretive language, and its focus on supporting reachability analysis. It's important to note that INTLAB [32], a closed-source interval arithmetic library, was not included in our comparison. Benchmarking CORA against INTLAB can be found in [5]. All tests were conducted within the identical physical environment as described in Sect. 3. The time consumption for each operation was measured by averaging the processing time for $N = 10^4$ sets of data randomly sampled from uniform distributions. The interval data used in the tests were defined as $[I, I + I_\delta]$, where I and I_δ are sampled from intervals $[\underline{I}, \overline{I}]$ and $[\underline{I}_\delta, \overline{I}_\delta]$, respectively. Both PyBDR and CORA utilized the double-precision data type compliant with the IEEE 754 standard [41]. The experimental settings and test results for all supported interval arithmetic operations by Interval

Tensor in PyBDR are summarized in Table 2, with the maximum relative error ϵ for each test defined as:

$$\epsilon = \max(\mu_1, \ldots, \mu_N), \quad \mu_j = \frac{\max|\underline{I}_{P,j} - \underline{I}_{C,j}|, |\overline{I}_{P,j} - \overline{I}_{C,j}|}{\overline{I}_{P,j} - \underline{I}_{P,j}} \tag{1}$$

where $[\underline{I}_{P,j}, \overline{I}_{P,j}]$ and $[\underline{I}_{C,j}, \overline{I}_{C,j}]$ refer to the bounds for the j^{th} test in PyBDR and CORA, respectively.

Symbolic Derivatives. Many continuous dynamical systems are typically described by Ordinary Differential Equations (ODEs), which depict their evolution within a state space [20,40]. Higher-order derivatives are frequently employed to provide more accurate approximations of system behaviors within local state neighborhoods. These derivatives often manifest as high-dimensional data structures. For instance, for a vector-valued function $\boldsymbol{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the second-order derivatives of $\boldsymbol{f}$ involve tensors of size $m \times n \times n$. As many set-based reachability analysis algorithms strive to approximate system behaviors, computing derivatives at specific time points becomes crucial for accurate approximations. Higher-order derivatives play a significant role in this process, allowing for more accurate approximations within local state neighborhoods. However, the computational overhead associated with calculating derivatives increases exponentially with their order, necessitating careful consideration in practical implementations. Unlike real-valued derivative evaluations, reachability analysis algorithms often rely on interval arithmetic. This approach is essential for estimating bounds that encompass exact values, accommodating potential errors inherent in real-world systems, and ensuring rigorous formal guarantees in analysis. Existing reachability analysis tools typically provide limited data structures for managing these complex operations efficiently. This limitation can lead to challenges in implementing theoretically straightforward operations, introducing unnecessary complexity and potentially compromising code readability and maintainability.

In response to the computational challenges posed by reachability analysis and the limitations of existing tools, our toolkit PyBDR integrates SymPy [26], providing a streamlined interface for evaluating and differentiating vector-valued functions effortlessly. This integration allows for precise handling of higher-order derivatives essential for accurate system behavior approximation. Moreover, our methodology leverages the interval tensor discussed earlier, enabling evaluations and derivatives within the framework of interval arithmetic, thereby enhancing operational convenience and adaptability. To assess the effectiveness of our approach, we provide detailed performance evaluations in Table 3 when handling data of varying scales across different systems.

Table 3. Performance evaluation of derivative computations in PyBDR.

System	Dimension			Mode	Run	Order	Avg. Time [s]	
	Input X	Input U	Output				w.r.t. X	w.r.t. U
ltv [16]	3	4	3	REL	1	0	$5.58e^{-3}$	$1.83e^{-3}$
					2^+	0	$7.56e^{-6}$	$5.33e^{-6}$
					1	1	$1.02e^{-2}$	$2.61e^{-3}$
					2^+	1	$5.42e^{-6}$	$4.63e^{-6}$
					1	3	$3.03e^{-2}$	$6.21e^{-2}$
					2^+	3	$1.17e^{-5}$	$1.83e^{-5}$
				INT	1	0	$2.26e^{-3}$	$2.05e^{-3}$
					2^+	0	$2.01e^{-4}$	$2.30e^{-4}$
					1	1	$2.47e^{-3}$	$9.17e^{-4}$
					2^+	1	$1.56e^{-4}$	$2.75e^{-5}$
					1	3	$1.48e^{-2}$	$2.38e^{-2}$
					2^+	3	$1.64e^{-4}$	$2.07e^{-4}$
Tank6eq [7]	6	1	6	REL	1	0	$8.87e^{-3}$	$3.49e^{-3}$
					2^+	0	$1.06e^{-5}$	$1.04e^{-5}$
					1	1	$2.81e^{-2}$	$1.82e^{-3}$
					2^+	1	$1.10e^{-5}$	$4.84e^{-6}$
					1	3	$3.56e^{-1}$	$4.44e^{-3}$
					2^+	3	$9.55e^{-5}$	$6.22e^{-6}$
				INT	1	0	$4.94e^{-3}$	$4.97e^{-3}$
					2^+	0	$3.98e^{-4}$	$4.35e^{-4}$
					1	1	$6.70e^{-3}$	$6.80e^{-4}$
					2^+	1	$5.31e^{-4}$	$2.19e^{-5}$
					1	3	$1.68e^{-1}$	$2.40e^{-3}$
					2^+	3	$2.03e^{-3}$	$3.37e^{-5}$
Quadrocopter [10]	12	3	12	REL	1	0	$1.49e^{-2}$	$9.01e^{-3}$
					2^+	0	$2.57e^{-5}$	$2.66e^{-5}$
					1	1	$9.49e^{-2}$	$6.49e^{-3}$
					2^+	1	$6.72e^{-5}$	$7.16e^{-6}$
					1	3	5.28	$1.05e^{-1}$
					2^+	3	$4.01e^{-3}$	$3.36e^{-5}$
				INT	1	0	$1.76e^{-2}$	$1.48e^{-2}$
					2^+	0	$2.51e^{-3}$	$2.51e^{-3}$
					1	1	$2.40e^{-2}$	$3.10e^{-3}$
					2^+	1	$5.69e^{-3}$	$6.03e^{-5}$
					1	3	2.67	$4.98e^{-2}$
					2^+	3	$7.29e^{-2}$	$3.95e^{-4}$
Lac Operon [15]	2	0	2	REL	1	0	$9.66e^{-3}$	–
					2^+	0	$7.61e^{-6}$	–
					1	1	$4.71e^{-2}$	–
					2^+	1	$1.44e^{-5}$	–
					1	3	2.82	–
					2^+	3	$1.09e^{-4}$	–
				INT	1	0	$5.80e^{-3}$	–
					2^+	0	$4.32e^{-4}$	–
					1	1	$1.88e^{-2}$	–
					2^+	1	$1.81e^{-3}$	–
					1	3	$1.02e^{-1}$	–
					2^+	3	$1.63e^{-2}$	–

Note: X – states of the systems; U – control inputs of the systems; INT – interval arithmetic; REL – real number arithmetic; 2^+ – second run and all subsequent runs.

3 Evaluation

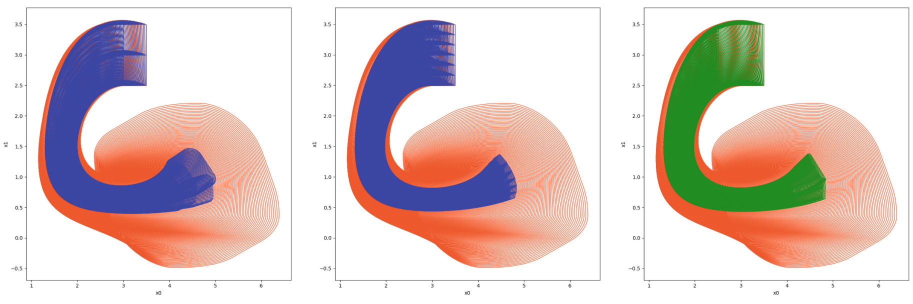
To illustrate the advancements facilitated by the set-boundary propagation technique implemented in PyBDR for reachability analysis, we conducted two sets of case studies. In the first category of case studies, we compared the performance of PyBDR when computing reachable sets using the set-boundary propagation technique against a baseline method employing a simple partitioning on the entire initial set. This comparison aimed to demonstrate the efficiency gains and accuracy improvements achieved through the set-boundary propagation technique. In the second category of case studies, we benchmarked PyBDR against CORA, a tool developed in MATLAB that also utilizes set propagation techniques for reachability analysis. This benchmarking focused on scenarios involving large initial sets and long time horizons, specifically in the context of safety verification for nonlinear systems. We ensured experimental fairness and parameter consistency by employing conservative linearization method [7] across all computations.

All experiments were performed on a Windows system equipped with an i7-13700H 2.1 GHz CPU with 32 GB RAM. Parallel operations were performed using 4 cores.

3.1 Comparative Studies on the Use of Boundary Analysis

Consider a Lotka-Volterra model of 2 variables [15] as follows,

$$\dot{x_0} = 1.5x_0 - x_0x_1 \tag{2}$$
$$\dot{x_1} = -3x_1 + x_0x_1 \tag{3}$$



(a) N=4; T=19.12. (b) N=36; T=148.73. (c) N=8; T=25.59.

Fig. 4. Reachable sets via simple partition (blue), boundary analysis (green), and baseline method without partition or boundary analysis (orange); N–number of cells, T–runtime in seconds. (Color figure online)

When starting with an initial set $[2.5, 3.5] \times [2.5, 3.5]$ and step size 0.005, the reachable set over time horizon $[0, 2.2]$ using different levels of partitioning over the initial set and based on boundary analysis is illustrated in Fig. 4. It is evident that as the simple partitioning method is applied to the initial set with increasing precision, smaller subsets are used for reachability analysis. This reduction in volume effectively reduces the error introduced by the wrapping effect, thereby mitigating the divergence of the reachable set over the specified time horizon. In contrast, the set-boundary propagation technique achieves a comparable improvement in the conservatism of reachability analysis using a limited number of cells that specifically enclose the boundary of the initial set. This approach provides a computationally efficient alternative to simple partitioning, demonstrating its effectiveness in advancing reachability analysis methods.

3.2 Comparative Studies on Reachability Analysis

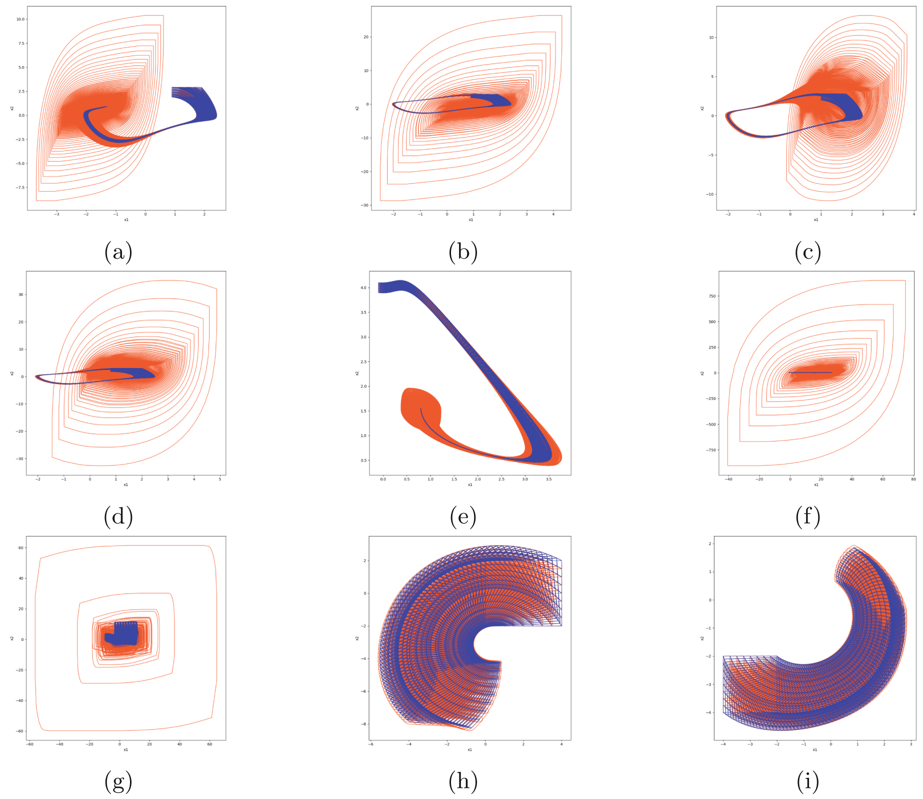


Fig. 5. Reachable sets obtained with CORA (orange) and PyBDR (blue). (Color figure online)

We benchmarked our performance against CORA on various continuous nonlinear dynamic systems, including Neural ODEs (NODEs), as listed in Table 4. Similarly, we applied the simple partition technique as in Subsect. 3.1 to improve the performance of CORA in reachability analysis. In our setup, PyBDR runs on 4 cores in parallel for specific operations, while CORA is single-threaded by its design. The runtimes in Table 4 refer to wall time, including I/O and other overheads, to compare overall performance in reachable set computation. For

Table 4. Reachability analysis comparison on continuous benchmarks

System	δ	$\mathcal{X}_0$	**T**	ϵ	**Fig.**	**PyBDR**		**CORA**	
						Cells	Time [s]	Cells	Time [s]
Vanderpol	0.01	$[0.9, 1.9] \times [1.9, 2.9]$	5.00	0.50	—	2×4	N/A	2×2	N/A
				0.33	—	3×4	44.80	3×3	N/A
				0.25	5a	4×4	47.98	4×4	44.30
		$[0.9, 1.9] \times [1.9, 2.9]$	9.00	0.20	—	5×4	N/A	5×5	N/A
				0.17	—	6×4	193.24	6×6	N/A
				0.14	—	7×4	237.23	7×7	N/A
				0.12	—	8×4	239.25	8×8	N/A
				0.11	5b	9×4	243.09	9×9	614.03
		$[1.0, 1.8] \times [2.0, 2.8]$	7.00	0.27	—	3×4	N/A	3×3	N/A
				0.20	—	4×4	108.98	4×4	N/A
				0.16	5c	5×4	114.80	5×5	174.66
		$[0.8, 2.0] \times [1.8, 3.0]$	7.00	0.24	—	5×4	N/A	5×5	N/A
				0.20	—	6×4	155.25	6×6	N/A
				0.17	—	7×4	164.23	7×7	N/A
				0.15	—	8×4	180.60	8×8	N/A
				0.13	5d	9×4	183.96	9×9	470.03
Brusselator [15]	0.01	$[-0.1, 0.1] \times [3.9, 4.1]$	5.00	0.07	—	3×4	N/A	3×3	N/A
				0.05	—	4×4	81.70	4×4	N/A
				0.04	5e	5×4	91.96	5×5	166.42
Synchronous Machine [35]	0.01	$[-0.7, 0.7] \times [2.3, 3.7]$	7.00	0.33	—	3×4	N/A	3×3	N/A
				0.25	—	4×4	99.73	4×4	N/A
				0.20	—	5×4	153.37	5×5	N/A
				0.17	5f	6×4	159.76	6×6	262.63
Lorenz [15]	0.02	$[-11, 3] \times [-3, 11] \times [-3, 11]$	1.00	2.33	—	$6 \times 6 \times 6$	N/A	$6 \times 6 \times 6$	N/A
				2.00	—	$7 \times 7 \times 6$	356.75	$7 \times 7 \times 7$	N/A
				1.75	5g	$8 \times 8 \times 6$	439.95	$8 \times 8 \times 8$	416.92
(NODE) Spiral 1 [25]	0.1	$[0, 4] \times [-2, 2]$	7.00	1.00	—	4×4	N/A	4×4	N/A
				0.80	—	5×4	857.26	5×5	N/A
				0.67	—	6×4	914.38	6×6	N/A
				0.57	—	7×4	1009.73	7×7	N/A
				0.50	5h	8×4	1094.89	8×8	1080.92
(NODE) Spiral 2 [25]	0.1	$[-4, -2] \times [-4, -2]$	7.00	0.50	—	4×4	N/A	4×4	N/A
				0.40	—	5×4	723.42	5×5	N/A
				0.33	—	6×4	885.98	6×6	N/A
				0.29	—	7×4	1121.64	7×7	N/A
				0.25	5i	8×4	1258.13	8×8	1549.61

Note: N/A – set explosion; δ – step; $\mathcal{X}_0$ – initial set; T – time horizon $[0, T]$; ϵ – max width of cell; Fig. – subfigure index in Fig 4.

each system, we present an initial setup that can lead to a set explosion due to the wrapping effect during computation. On this basis, we reduce the conservativeness of the reachable set by using a more refined boundary characterization in PyBDR, and by partitioning the initial set into smaller cells in CORA. It is noteworthy that since the boundary of sets is dimensionally degenerate relative to the sets themselves, we constrained cell's maximum width in both methods to keep the error introduced by the wrapping effect for each cell within the same scale.

In Table 4, we observe that for systems with relatively large initial sets and long time horizons, both PyBDR and CORA suffer from significant errors from the wrapping effect, which leads to an overestimation of reachable sets. By reducing cell size, both tools yield more accurate over-approximations of reachable sets within specified time horizons. Moreover, as shown in Fig. 4, we can always obtain a more accurate estimation. Notably, despite Python's inherent limitations in iterative computations when compared to MATLAB, by processing each cell in parallel, our toolkit still significantly outperforms CORA in terms of overall computation time, as particularly evidenced by the results presented in Fig. 5d. In particular, the analysis of the VanderPol system with an initial set $[0.9, 1.9] \times [1.9, 2.9]$ indicates a requirement for finer cell granularity to accurately approximate the reachable set as the time horizon extends. This refinement leads to a pronounced increase in computational time for CORA compared to PyBDR. And this trend persists across different initial set within $[0, 7]$, where the need for precision intensifies to maintain valid reachable set estimations.

4 Conclusion

In this paper, we presented PyBDR, a Python-based toolkit that enhances the reachability analysis through set-boundary propagation analysis. Its key features include advanced set-boundary analysis to mitigate the wrapping effect and the integration of tensor-level interval arithmetic for efficient computations. Besides, PyBDR offers a diverse range of set representations and supports symbolic computation of derivatives, crucial for precise system behavior analysis. Built with Python's user-friendly environment in mind, PyBDR facilitates rapid prototyping and accommodates complex computational tasks effectively. Its capabilities are demonstrated through benchmarking across various nonlinear dynamics scenarios.

For future development, our focus will expand to include support for additional dynamical systems, particularly hybrid systems. We also plan to incorporate a broader array of set representations, including nonconvex forms such as polynomial zonotopes. Enhancing user interaction through a user-friendly and interactive visualization module is another pivotal aspect of our roadmap.

Acknowledgement. This work is funded by the CAS Pioneer Hundred Talents Program and Basic Research Program of Institute of Software, CAS (Grant No. ISCAS-JCMS-202302).

Data Availability Statement. The artifact for this work is available at https://doi.org/10.5281/zenodo.12206996, and PyBDR is available at https://github.com/ASAG-ISCAS/PyBDR.

References

1. Alanwar, A., Said, H., Althoff, M.: Distributed secure state estimation using diffusion Kalman filters and reachability analysis. In: 2019 IEEE 58th Conference on Decision and Control (CDC), pp. 4133–4139. IEEE (2019)
2. Althoff, M.: Reachability analysis of nonlinear systems using conservative polynomialization and non-convex sets. In: Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control, pp. 173–182 (2013)
3. Althoff, M.: An introduction to CORA 2015. In: Proceedings of the 1st and 2nd Workshop on Applied Verification for Continuous and Hybrid Systems, pp. 120–151. EasyChair (2015). https://doi.org/10.29007/zbkv, https://easychair.org/publications/paper/xMm
4. Althoff, M., Frehse, G., Girard, A.: Set propagation techniques for reachability analysis. Ann. Rev. Control Robot. Auton. Syst. **4**, 369–395 (2021)
5. Althoff, M., Grebenyuk, D.: Implementation of interval arithmetic in CORA 2016. In: Proceedings of the 3rd International Workshop on Applied Verification for Continuous and Hybrid Systems, pp. 91–105 (2016)
6. Althoff, M., Rajhans, A., Krogh, B.H., Yaldiz, S., Li, X., Pileggi, L.: Formal verification of phase-locked loops using reachability analysis and continuization. Commun. ACM **56**(10), 97–104 (2013)
7. Althoff, M., Stursberg, O., Buss, M.: Reachability analysis of nonlinear systems with uncertain parameters using conservative linearization. In: 2008 47th IEEE Conference on Decision and Control, pp. 4042–4048. IEEE (2008)
8. Bak, S., Duggirala, P.S.: HyLAA: a tool for computing simulation-equivalent reachability for linear systems. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, pp. 173–178 (2017)
9. Bak, S., Duggirala, P.S.: Simulation-equivalent reachability of large linear systems with inputs. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 401–420. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_20
10. Beard, R.W.: Quadrotor dynamics and control. Brigham Young Univ. **19**(3), 46–56 (2008)
11. Bogomolov, S., Forets, M., Frehse, G., Potomkin, K., Schilling, C.: JuliaReach: a toolbox for set-based reachability. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 39–44 (2019)
12. Bogomolov, S., et al.: Assume-guarantee abstraction refinement meets hybrid systems. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 116–131. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13338-6_10
13. Chen, M., Tomlin, C.J.: Hamilton-Jacobi reachability: some recent theoretical advances and applications in unmanned airspace management. Ann. Rev. Control Robot. Auton. Syst. **1**, 333–358 (2018)
14. Chen, R.T., Rubanova, Y., Bettencourt, J., Duvenaud, D.K.: Neural ordinary differential equations. In: Advances in Neural Information Processing Systems, vol. 31 (2018)
15. Chen, X.: Reachability analysis of non-linear hybrid systems using taylor models. Ph.D. thesis, Fachgruppe Informatik, RWTH Aachen University (2015)

16. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
17. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
18. Granvilliers, L., Benhamou, F.: Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. ACM Trans. Math. Softw. (TOMS) **32**(1), 138–156 (2006)
19. Harris, C.R., et al.: Array programming with NumPy. Nature **585**(7825), 357–362 (2020). https://doi.org/10.1038/s41586-020-2649-2
20. Khalil, H.K.: Nonlinear Systems. Prentice Hall, Upper Saddle River (NJ), 3. ed., international ed. edn. (2000)
21. Kühn, W.: Zonotope dynamics in numerical quality control. In: Hege, HC., Polthier, K. (eds.) Mathematical Visualization: Algorithms, Applications and Numerics, pp. 125–134. Springer, Heidelberg (1998). https://doi.org/10.1007/978-3-662-03567-2_10
22. Le Guernic, C., Girard, A.: Reachability analysis of hybrid systems using support functions. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 540–554. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_40
23. Liang, Z., Ren, D., Liu, W., Wang, J., Yang, W., Xue, B.: Safety verification for neural networks based on set-boundary analysis. In: David, C., Sun, M. (eds.) Theoretical Aspects of Software Engineering, pp. 248–267. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-35257-7_15
24. Liu, E.I., Würsching, G., Klischat, M., Althoff, M.: CommonRoad-Reach: a toolbox for reachability analysis of automated vehicles. In: 2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC), pp. 2313–2320. IEEE (2022)
25. Manzanas Lopez, D., Musau, P., Hamilton, N.P., Johnson, T.T.: Reachability analysis of a general class of neural ordinary differential equations. In: Bogomolov, S., Parker, D. (eds.) Formal Modeling and Analysis of Timed Systems. FORMATS 2022. LNCS, vol. 13465. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15839-1_15
26. Meurer, A., et al.: SymPy: symbolic computing in Python. PeerJ Comput. Sci. **3**, e103 (2017). https://doi.org/10.7717/peerj-cs.103
27. Mitchell, I.M., Bayen, A.M., Tomlin, C.J.: A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. IEEE Trans. Autom. Control **50**(7), 947–957 (2005)
28. Moore, R.E.: Interval Analysis, vol. 4. Prentice-Hall Englewood Cliffs (1966)
29. Park, J., Özgüner, Ü.: Model based controller synthesis using reachability analysis that guarantees the safety of autonomous vehicles in a convoy. In: 2012 IEEE International Conference on Vehicular Electronics and Safety (ICVES 2012), pp. 134–139. IEEE (2012)
30. Ray, R., Gurung, A., Das, B., Bartocci, E., Bogomolov, S., Grosu, R.: XSpeed: accelerating reachability analysis on multi-core processors. In: Piterman, N. (ed.) HVC 2015. LNCS, vol. 9434, pp. 3–18. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26287-1_1

31. Ren, D., Liang, Z., Wu, C., Ding, J., Wu, T., Xue, B.: Inner-approximate reachability computation via zonotopic boundary analysis. In: To appear in Computer Aided Verification: 36th International Conference, CAV 2024 (2024)
32. Rump, S.M. (1999). INTLAB — INTerval LABoratory. In: Csendes, T. (eds) Developments in Reliable Computing. Springer, Dordrecht (1999). https://doi.org/10.1007/978-94-017-1247-7_7
33. Schupp, S., Ábrahám, E., Makhlouf, I.B., Kowalewski, S.: HyPro: A C++ library of state set representations for hybrid systems reachability analysis. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 288–294. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_20
34. Schürmann, B.: Using reachability analysis in controller synthesis for safety-critical systems. Ph.D. thesis, Technische Universität München (2022)
35. Susuki, Y., et al.: A hybrid system approach to the analysis and design of power grid dynamic performance. Proc. IEEE **100**(1), 225–239 (2011)
36. Tang, C., Althoff, M.: Formal verification of robotic contact tasks via reachability analysis. IFAC-PapersOnLine **56**(2), 7912–7919 (2023)
37. Xue, B., Easwaran, A., Cho, N.J., Fränzle, M.: Reach-avoid verification for nonlinear systems based on boundary analysis. IEEE Trans. Autom. Control **62**(7), 3518–3523 (2016)
38. Xue, B., She, Z., Easwaran, A.: Under-approximating backward reachable sets by polytopes. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 457–476. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_25
39. Xue, B., Wang, Q., Feng, S., Zhan, N.: Over-and underapproximating reach sets for perturbed delay differential equations. IEEE Trans. Autom. Control **66**(1), 283–290 (2020)
40. Yang, B., Stipanovic, D.: Nonlinear Systems: Recent Developments and Advances (2023)
41. Zuras, D., et al.: IEEE standard for floating-point arithmetic. IEEE Std. **754**(2008), 1–70 (2008)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Discourje: Run-Time Verification of Communication Protocols in Clojure — Live at Last

Sung-Shik Jongmans(✉)

Open University of the Netherlands, Heerlen, The Netherlands
ssj@ou.nl

FM Artifact Evaluation ★★ Functional

Abstract. Multiparty session typing (MPST) is a formal method to make concurrent programming simpler. The idea is to use type checking to automatically prove safety (protocol compliance) and liveness (communication deadlock freedom) of implementations relative to specifications. Discourje is an existing run-time verification library for communication protocols in Clojure, based on dynamic MPST. The original version of Discourje can detect only safety violations. In this paper, we present an extension of Discourje to detect also liveness violations.

1 Introduction

Background. With the advent of multicore processors, multithreaded programming—a notoriously error-prone enterprise—has become increasingly important.

Because of this, mainstream languages have started to offer core support for higher-level *communication* primitives besides lower-level *synchronisation* primitives (e.g., Clojure, Go, Kotlin, Rust). The idea has been to add *message passing* as an abstraction on top of *shared memory*, for—supposedly—*channels* are easier to use than *locks*. However, empirical research shows that, actually, "message passing does not necessarily make multithreaded programs less error-prone than shared memory" [36]. One of the core challenges is as follows: given a specification S of the *communication protocols* that an implementation I should fulfil, how to prove that I is *safe* and *live* relative to S? Safety means that "bad" channel actions never happen: if a channel action happens in I, then it is allowed to happen by S (protocol compliance). Liveness means that "good" channel actions eventually happen (communication deadlock freedom).

Multiparty Session Typing (MPST). MPST [17] is a formal method to automatically prove safety and liveness of implementations relative to specifications. The idea is to implement communication protocols as *sessions* (of communicating threads), specify them as *behavioural types* [1,21], and verify the former against the latter using behavioural type checking. Formally, the central theorem is that well-typedness implies safety and liveness. Over the past fifteen

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 158–166, 2025.
https://doi.org/10.1007/978-3-031-71177-0_11

years, much progress has been made, including the development of many tools to combine MPST with mainstream languages (e.g., F# [31], F* [37], Go [9], Java [19,20], OCaml [22], Rust [26,27], Scala [3,10,11,34], and TypeScript [29]).

Behavioural type checking can be done *statically* at compile-time or *dynamically* at run-time. The disadvantage of static MPST is, it is conservative: statically checking *each possible run* of a session is often prohibitively complicated—if computable at all—so sessions are often unnecessarily rejected. In contrast, the advantage of dynamic MPST is, it is liberal: dynamically checking *one actual run* of a session is much simpler, so sessions are never unnecessarily rejected.

This Work. *Discourje* (pronounced "discourse") [13,14,18] is a library that adds dynamic MPST to *Clojure*[1]. It has a specification language to write behavioural types (embedded as an internal DSL in Clojure) and a verification engine to dynamically type-check sessions against them. The **key design goals** have been to achieve high expressiveness (cf. static MPST) and to be particularly mindful of ergonomics (i.e., make Discourje's usage as frictionless as possible).

In a nutshell, at run-time, Discourje's dynamic type checker simulates behavioural type S—as if it were a state machine—alongside session I. Each time when a channel action is about to happen in I, the dynamic type checker intervenes and first verifies if a corresponding transition can happen in S. If so, both the channel action and the transition happen. If not, an exception is thrown.

However, while safety violations are detected in this way (protocol incompliance), liveness violations are not (communication deadlocks: threads cyclically depend on each others' channel actions, and so, they collectively get stuck). This is a serious limitation relative to static MPST. In this paper, we present an extension of Discourje to detect also liveness violations. Achieving this, *without compromising the key design goals*, has been an elusive problem that for years we did not know how to solve (e.g., we could not reuse variants of existing techniques for static MPST at run-time, as this would negatively affect expressiveness).

Section 2 of this paper demonstrates <u>that</u> it can be done, while Sect. 3 outlines <u>how</u>. The key idea is to use "mock" channels, which mimic "real" channels, to track ongoing communications: before any channel action happens on a real channel, it is first tried on a corresponding mock channel, allowing us to check if *all* threads would get stuck in a *total* communication deadlock as a result.

2 Demonstration

We demonstrate the extension to detect liveness violations with two examples. For reference, Fig. 1 summarises the main elements of Discourje and Clojure.

Example 1. The *Two-Buyer* protocol consists of *Buyer1*, *Buyer2*, and *Seller* [17]: "Buyer1 and Buyer2 wish to buy an expensive book from Seller by combining

[1] A Lisp that runs on the JVM, with core support for channel-based message passing.

their money. Buyer1 sends the title of the book to Seller, Seller sends to both Buyer1 and Buyer2 its quote, Buyer1 tells Buyer2 how much she can pay, and Buyer2 either accepts the quote or rejects the quote by notifying Seller."

Figure 2 shows a behavioural type and a session. It is safe and live. In contrast, if we had accidentally written `(<!! c3)` on line 11 (i.e., Buyer1 tries to receive from Buyer2 instead of Seller), then it deadlocks. The original Discourje does not detect this liveness violation, but with the extension, an exception is thrown. □

Discourje:

– `(defthread` *id*`)`/`(defsession` *id* `[`*args*`]` *body*`)` specifies a thread name/protocol.
– `(-->>`/`-->` *t p q*`)` specifies an asynchronous/synchronous communication of a value of data type *t* through a buffered/unbuffered channel from *p* to *q*.
– `(alt ...)` and `(cat`/`par ...)` specify choice and sequencing/interleaving.
– Names of threads and protocols are prefixed by an otherwise meaningless colon.

Clojure:

– `(thread` *body*`)`, `(chan)`, and `(chan` *size*`)` implement the creation of new thread, a new unbuffered channel, and a new buffered channel.
– `(>!!` *ch expr*`)` implements the send of the value of *expr* through *ch*.
– `(<!!` *ch*`)` implements the receive of a value through *ch*.
– `(alts!! [`act_1 `...` act_n`])` implements a selection of one of the channel actions, depending on their dis/enabledness (cf. `select` of POSIX sockets and Go channels). If act_i is a send, it is a pair `[`*ch v*`]`; if it is a receive, it is just *ch*. The function returns a pair `[`*v ch*`]` where *v* is the value sent/received, and *ch* is the channel.

Fig. 1. Discourje and Clojure in a nutshell

```
 1 (defthread :buyer1)
 2 (defthread :buyer2)
 3 (defthread :seller)
 4
 5 (defsession :two-buyer []
 6   (cat
 7    (-->> String :buyer1 :seller)
 8    (par
 9     (cat
10      (-->> Double :seller :buyer1)
11      (-->> Double :buyer1 :buyer2))
12     (-->> Double :seller :buyer2))
13    (-->> Boolean :buyer2 :seller)))
```

(a) Specification in Discourje

```
 1 (def c1 (chan 1))
 2 (def c2 (chan 1))
 3 (def c3 (chan 1))
 4 (def c4 (chan 1))
 5 (def c5 (chan 1))
 6 (def c6 (chan 1))
 7
 8 (thread ;; Buyer1
 9   (>!! c1 "book")
10   (let
11     [x (<!! c5)
12      y (/ x 2)]
13     (>!! c2 y)))
```

```
14 (thread ;; Buyer2
15   (let
16     [x (<!! c6)
17      y (<!! c2)
18      z (= x y)]
19     (>!! c4 z)))
20
21 (thread ;; Seller
22   (<!! c1)
23   (>!! c5 20.00)
24   (>!! c6 20.00)
25   (println
26     (<!! c4)))
```

(b) Implementation in Clojure

To dynamically type-check the session, the following code creates a *monitor* for the session, and *links* it to each channel along with the intended sender and receiver:

```
(def m (monitor :two-buyer :n 3))
(link c1 :buyer1 :buyer2 m) (link c2 :buyer1 :seller m)
(link c3 :buyer2 :buyer1 m) (link c4 :buyer2 :seller m)
(link c5 :seller :buyer1 m) (link c6 :seller :buyer2 m)
```

Fig. 2. Two-Buyer (Example 1)

```
 1 (defthread :c) (defthread :s1)
 2 (defthread :b) (defthread :s2)
 3
 4 (defsession :load-balancer []
 5   (cat
 6    (-->> Long :c :b)
 7    (alt
 8     (cat
 9      (-->> Long :b  :s1)
10      (-->  Long :s1 :c))
11     (cat
12      (-->> Long :b  :s2)
13      (-->  Long :s2 :c)))))
```

(a) Specification in Discourje

```
 1 (def c1 (chan))
 2 (def c2 (chan))
 3 (def c3 (chan))
 4 (def c4 (chan 512))
 5 (def c5 (chan 1024))
 6
 7 ;; Load Balancer
 8 (thread
 9   (let [x (<!! c1)]
10     (alts!!
11       [[c4 x]
12        [c5 x]])))
13
```

```
14 (thread ;; Client
15   (>!! c1 5)
16   (alts!! [c2 c3]))
17
18 (thread ;; Server1
19   (let [x (<!! c2)
20         y (inc x)]
21     (>!! c2 y))))
22
23 (thread ;; Server2
24   (let [x (<!! c3)
25         y (inc x)]
26     (>!! c3 y))))
```

(b) Implementation in Clojure

To dynamically type-check the session:

```
(def m (monitor :load-balancer :n 4)) (link c4 :b :s1 m) (link c2 :s1 :c m)
                          (link c1 :c :b m) (link c5 :b :s2 m) (link c3 :s2 :c m)
```

Fig. 3. Load Balancing (Example 2)

Example 2. The *Load Balancing* protocol consists of *Client*, *Server1*, *Server2*, and *LoadBalancer*. First, a request is communicated synchronously from Client to LoadBalancer, and asynchronously from LoadBalancer to Server1 or Server2. Next, the response is communicated synchronously from that server to Client.

Figure 3 shows a behavioural type and a session. It is safe but not live. There are two deadlocks. The first one occurs because Server1 and Server2 try to receive from `c2` and `c3` on lines 19 and 23; this should be `c4` and `c5`. The second deadlock occurs because one of the servers will never receive a value and, as a result, block the entire program from terminating. The original Discourje does not detect these liveness violations, but with the extension, exceptions are thrown. □

3 Technical Details

Requirements. In this section, we outline how the extension to detect liveness violations works, focussing on the core deadlock detection algorithm. We begin by stating the rather complicated requirements for this algorithm, as entailed by Discourje's key design goals regarding expressiveness and ergonomics (Sect. 1):

- **Expressiveness:** The algorithm must be applicable to any combination of buffered and unbuffered channels, and to all functions `>!!` (send), `<!!` (receive), and `alts!!` (select). Thus, the programmer can continue to freely mix synchronous and asynchronous sends/receives, possibly selected dynamically.
- **Ergonomics:** The algorithm must call only into the public API of Clojure's standard libraries, without modifying the internals, and without relying on JVM interoperability. Thus, the programmer can write portable code that runs on different versions of Clojure and on different architectures.

The combination of these requirements has made the design of the algorithm elusive. For instance, the expressiveness requirement means that we cannot simply reuse existing distributed algorithms for deadlock detection (e.g., [6,16,25,35]), as they typically do not support mixing of synchrony and asynchrony. The ergonomics requirement means that we cannot instrument Clojure's internal code to manage threads, nor can we use Java's thread monitoring facilities.

Terminology. A *channel action* is either a *send* of v through ch, represented as `[`$ch\ v$`]`, or a *receive* through channel ch, represented as just ch (cf. `alts!!` in Fig. 1). A channel action is *pending* if it has been initiated but not yet completed. A pending channel action is either *enabled* or *disabled*, depending on ch:

– when ch is a buffered channel, a pending send is enabled iff ch is non-full, while a pending receive is enabled iff ch is non-empty;
– when ch is an unbuffered channel, a pending send is enabled iff a corresponding receive is pending, and vice versa.

When a thread initiates channel actions, but they are disabled, it is *suspended*. When a disabled channel action becomes enabled, the suspended thread is *resumed*. A *communication deadlock* is a situation where each thread is suspended.

Setting the Stage. Normally, channel actions are initiated via functions `>!!`, `<!!`, and `alts!!`. When these functions are called using the extension, the dynamic type checker intervenes and first calls `(detect-deadlocks [`$act_1 \ldots act_n$`])` to initiate corresponding "mock" channel actions on "mock" channels. Each mock channel mimics a "real" channel and is used only by the dynamic type checker.

The mock channels have the same un/buffered properties and contents as the real channels, except that values are replaced with tokens. So, if `detect-deadlocks` detects a deadlock on the mock channels, then a deadlock will occur on the real channels, too. (Mock channels are also essential to detect safety violations.)

To initiate the mock channel actions, a separate function in the public API of Clojure's standard libraries is used: `(do-alts`$f\ acts\ config$`)`. It resembles `alts!!`, except that it never suspends the calling thread. Instead, a call of `do-alts` immediately returns and, asynchronously, initiates the channel actions in *acts* and calls f when one is completed. In this way, initiation of mock channel actions can be decoupled from suspension of threads (demonstrated below).

Algorithm. Let `n` be the number of threads. The idea to detect deadlocks is to identify the situation when `n-1` threads are already suspended, while the **last thread** is **about to be suspended**. In that situation, instead of suspending the last thread, an exception is thrown to flag the liveness violation. In code:

```
(defn detect-deadlocks [mock-acts] ;; act1 ... actn
  (let [ret (about-to-be-suspended? mock-acts)]
    (if (true? ret)
      (let [ret (last-thread? mock-acts)]
        (if (true? ret) (throw (ex-info "deadlock!" {})) ret)) ret)))
```

Function about-to-be-suspended? checks if any of the mock-acts is enabled. If so, it immediately initiates and completes it, and returns the result (of the form [v ch]). If not, the function returns true to indicate that the current thread would indeed be suspended if mock-acts were to be initiated. In code:

```
6 (defn about-to-be-suspended? [mock-acts]
7   (let [ret @(do-alts (fn [_] nil) mock-acts {:default nil})]
8     (if (not= ret [nil :default]) ret true)))
```

On line 7, optional parameter {:default nil} configures alts!! such that it immediately returns [nil :default] when all mock-acts are disabled.

Function last-thread? increments the number of suspended threads and checks if the number is less than n. If so, it initiates mock-acts, and actually suspends the current thread. If not, the function returns true to indicate that the current thread is indeed the last one, so a deadlock is detected. In code:

```
9  (def i (atom 0)) ;; number of suspended threads (private to the algorithm)
10
11 (defn last-thread? [mock-acts]
12   (if (< (swap! i inc) n)   ;; increment `i` (`swap!` returns the new value)
13     (let [p (promise)]      ;; create promise to store result of `mock-acts`
14       (do-alts (fn [x] (deliver p x)) mock-acts {}) ;; initiate `mock-acts`,
15                             ;; and store result `x` of one of them in `p`
16                             ;; upon completion, all asynchronously
17       (let [ret (deref p)] ;; suspend thread (`deref` blocks until `deliver`)
18         (swap! i dec)       ;; decrement `i`
19         ret)
20     true))
```

The code shown so far explains the general idea behind the algorithm. However, the details are more involved: our presentation does not yet account for data races, several of which are possible. For instance, suppose that there are two threads (Alice and Bob), that they initiate corresponding channel actions (no deadlock), and that calls of detect-deadlocks are scheduled as follows:

> **(1)** Alice executes about-to-be-suspended?. It returns true. **(2)** Bob executes about-to-be-suspended?. It, again, returns true, as Alice has not yet executed last-thread?. **(3)** Bob executes last-thread?. It increments n to 1 and suspends Bob. **(4)** Alice executes last-thread?. It increments n to 2, detects that Alice is last, and immediately returns nil.

At this point, mistakenly, an exception is thrown. There are more subtle data races, too. The core issue is that about-to-be-suspended? and last-thread? should be run *atomically* to avoid problematic schedules (e.g., the one above). Details appear in the technical report [23, Sect. A]. The actual source code was validated using both unit tests and whole-program tests.

4 Conclusion

Closest to the work in this paper is existing work on dynamic MPST [4,15,30–32] and alternate forms of dynamic behavioural typing [7,8,12,28]. However, none of these tools can check for liveness at run-time. Also closely related is existing work on dynamic deadlock detection in distributed systems (e.g., [6,16,25,35]). However, as stated in Sect. 3, these algorithms do not fit our requirements. Finally, we are aware of two other works that use formal techniques to reason about Clojure programs: the formalisation of an optional type system for Clojure [5], and a translation from Clojure to Boogie [2,33]. In future work, we aim to study and optimise the performance overhead of our deadlock detection algorithm.

Data Availability Statement. The artifact is available on Zenodo [24]. It contains the new version of Discourje, including the examples of this paper.

Disclosure of Interests. The author has no competing interests to declare that are relevant to the content of this article.

References

1. Ancona, D., et al.: Behavioral types in programming languages. Found. Trends Program. Lang. **3**(2–3), 95–230 (2016)
2. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: FMCO. LNCS, vol. 4111 (2005)
3. Barwell, A.D., Hou, P., Yoshida, N., Zhou, F.: Designing asynchronous multiparty protocols with crash-stop failures. In: ECOOP. LIPIcs, vol. 263, pp. 1:1–1:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023)
4. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. Theor. Comput. Sci. **669**, 33–58 (2017)
5. Bonnaire-Sergeant, A., Davies, R., Tobin-Hochstadt, S.: Practical optional types for Clojure. In: ESOP. LNCS, vol. 9632 (2016)
6. Bracha, G., Toueg, S.: Distributed deadlock detection. Distributed Comput. **2**(3), 127–138 (1987)
7. Burlò, C.B., Francalanza, A., Scalas, A.: On the monitorability of session types, in theory and practice. In: ECOOP. LIPIcs, vol. 194, pp. 20:1–20:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
8. Burlò, C.B., Francalanza, A., Scalas, A., Trubiani, C., Tuosto, E.: PSTMonitor: monitor synthesis from probabilistic session types. Sci. Comput. Program. **222**, 102847 (2022)
9. Castro, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in Go: statically-typed endpoint APIs for dynamically-instantiated communication structures. PACMPL **3**(POPL), 1–30 (2019)
10. Cledou, G., Edixhoven, L., Jongmans, S., Proença, J.: API generation for multiparty session types, revisited and revised using Scala 3. In: ECOOP. LIPIcs, vol. 222 (2022)
11. Ferreira, F., Jongmans, S.: Oven: Safe and live communication protocols in Scala, using synthetic behavioural type analysis. In: ISSTA, pp. 1511–1514. ACM (2023)
12. Gommerstadt, H., Jia, L., Pfenning, F.: Session-typed concurrent contracts. J. Log. Algebraic Methods Program. **124**, 100731 (2022)
13. Hamers, R., Jongmans, S.: Discourje: runtime verification of communication protocols in Clojure. In: TACAS (1). LNCS, vol. 12078 (2020)

14. Hamers, R., Jongmans, S.-S.: Safe sessions of channel actions in clojure: a tour of the discourje project. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12476, pp. 489–508. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_28
15. van den Heuvel, B., Pérez, J.A., Dobre, R.A.: Monitoring blackbox implementations of multiparty session protocols. In: RV. Lecture Notes in Computer Science, vol. 14245, pp. 66–85. Springer (2023). https://doi.org/10.1007/978-3-031-44267-4_4
16. Hilbrich, T., de Supinski, B.R., Nagel, W.E., Protze, J., Baier, C., Müller, M.S.: Distributed wait state tracking for runtime MPI deadlock detection. In: SC, pp. 16:1–16:12. ACM (2013)
17. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL (2008)
18. Horlings, E., Jongmans, S.: Analysis of specifications of multiparty sessions with dcj-lint. In: ESEC/SIGSOFT FSE, pp. 1590–1594. ACM (2021)
19. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: FASE. LNCS, vol. 9633 (2016)
20. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: FASE. LNCS, vol. 10202 (2017)
21. Hüttel, H., et al.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 1–36 (2016)
22. Imai, K., Neykova, R., Yoshida, N., Yuen, S.: Multiparty session programming with global protocol combinators. In: ECOOP. LIPIcs, vol. 166 (2020)
23. Jongmans, S.S.: Discourje: run-time verification of communication protocols in clojure – Live at last. Technical report (2023). https://arxiv.org/abs/2407.00540
24. Jongmans, S.: Discourje: run-time verification of communication protocols in Clojure – live at last (artifact) (2024). https://doi.org/10.5281/zenodo.12519843
25. Krivokapic, N., Kemper, A., Gudes, E.: Deadlock detection in distributed database systems: a new algorithm and a comparative performance analysis. VLDB J. **8**(2), 79–100 (1999)
26. Lagaillardie, N., Neykova, R., Yoshida, N.: Implementing multiparty session types in rust. In: COORDINATION. LNCS, vol. 12134 (2020)
27. Lagaillardie, N., Neykova, R., Yoshida, N.: Stay safe under panic: affine Rust programming with multiparty session types. In: ECOOP. LIPIcs, vol. 222 (2022)
28. Melgratti, H.C., Padovani, L.: Chaperone contracts for higher-order sessions. Proc. ACM Program. Lang. **1**(ICFP), 1–29 (2017)
29. Miu, A., Ferreira, F., Yoshida, N., Zhou, F.: Communication-safe web programming in TypeScript with routed multiparty session types. In: CC (2021)
30. Neykova, R., Bocchi, L., Yoshida, N.: Timed runtime monitoring for multiparty conversations. Formal Asp. Comput. **29**(5), 877–910 (2017)
31. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in F#. In: CC (2018)
32. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: CC (2017)
33. Pinzaru, G., Rivera, V.: Towards static verification of Clojure contract-based programs. In: TOOLS. LNCS, vol. 11771 (2019)
34. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP. LIPIcs, vol. 74 (2017)
35. Srinivasan, S., Rajaram, R.: A decentralized deadlock detection and resolution algorithm for generalized model in distributed systems. Distrib. Parallel Databases **29**(4), 261–276 (2011)

36. Tu, T., Liu, X., Song, L., Zhang, Y.: Understanding real-world concurrency bugs in Go. In: ASPLOS (2019)
37. Zhou, F., Ferreira, F., Hu, R., Neykova, R., Yoshida, N.: Statically verified refinements for multiparty protocols. Proc. ACM Program. Lang. **4**(OOPSLA), 1–30 (2020)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Stochastic Games for User Journeys

Paul Kobialka[1](✉), Andrea Pferscher[1], Gunnar R. Bergersen[1,2], Einar Broch Johnsen[1], and Silvia Lizeth Tapia Tarifa[1]

[1] University of Oslo, Oslo, Norway
{paulkob,andreapf,gunnab,einarj,sltarifa}@ifi.uio.no
[2] GrepS B.V., Utrecht, The Netherlands
gunnar.bergersen@greps.com

Abstract. Industry is shifting towards service-based business models, for which user satisfaction is crucial. User satisfaction can be analyzed with user journeys, which model services from the user's perspective. Today, these models are created manually and lack both formalization and tool-supported analysis. This limits their applicability to complex services with many users. Our goal is to overcome these limitations by automated model generation and formal analyses, enabling the analysis of user journeys for complex services and thousands of users. In this paper, we use stochastic games to model and analyze user journeys. Stochastic games can be automatically constructed from event logs and model checked to, e.g., identify interactions that most effectively help users reach their goal. Since the learned models may get large, we use property-preserving model reduction to visualize users' pain points to convey information to business stakeholders. The applicability of the proposed method is here demonstrated on two complementary case studies.

Keywords: User journeys · Data-driven model construction · Automata learning · Model checking · Stochastic games · PRISM

1 Introduction

The *servitization of business* describes a shift towards offering products as services [44]. This shift makes companies more dependent on user satisfaction; e.g., it has become much easier to change service providers. Investment in user satisfaction pays off [17], which raises the following question: How can we formally model and analyze the way users experience their interaction with a service?

User journeys model services from the users' perspective [41]. They describe how users employ a service to achieve a goal. User journeys may include many paths, capturing different sequences of actions between a service and its users. These models enable the analysis of user experience along different (intended or unintended) paths through a service. Although most user journeys today are

This work is part of the *Smart Journey Mining* project, funded by the Research Council of Norway (grant no. 312198).

© The Author(s) 2025

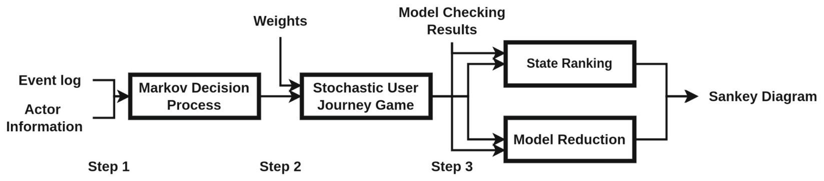


Fig. 1. Steps to create Sankey diagrams from the event logs of the case studies.

created manually by domain experts and the associated user experience is captured through interviews [22,41], the method has been successful at providing feedback to improve services. However, tool support for the modeling and analysis of user journeys is sparse [23], which makes the method difficult to apply in complex domains and to services with numerous and diverse users.

A recent line of work aims to automatically mine user journeys and analyze them using formal methods [26,28,30,31]. This significantly reduces the manual effort needed to create models and enables a different scale of complexity in the analyzed services and number of users. Starting from event logs, which are widely available for software services, *process mining* [1] and *automata learning* [18] can automatically generate behavioral models of user journeys from these logs, such as finite state automata. These can then be analyzed by *model checking* [4].

This paper goes beyond previous work by modeling user journeys as *stochastic games* [11]. We exploit the underlying distribution of events in the event log, which was ignored in previous work. Stochastic games allow complex user behavior to be captured, yet the resulting games can still be model checked. Figure 1 summarizes the steps applied to event logs to analyze user experience. These steps elegantly combine and extend several known techniques. *Step 1* generates stochastic automata from event logs by means of automata learning. *Step 2* converts the learned automata into stochastic weighted games. The resulting games are analyzed using probabilistic model checking to derive optimal strategies. *Step 3* ranks critical actions after which users tend to abandon their journey and visualizes the outcome of these novel analyses via a property-preserving visualization technique, to improve the interpretability of the stochastic game results.

We apply these steps to two case studies: an industrial case study [30,31] and a benchmark [15] from the literature. The case studies are complementary in complexity and differ in the number of users. In both cases, we identified potential service improvements and automatically uncovered caveats. The case studies suggest that our method is able to address two pressing industrial challenges: (1) the automated construction of stochastic user journey models for complex services from event logs, and (2) identification of service bottlenecks by automated analysis of models that reflect user experience. In short, the contributions of this paper are: (1) a formalization of user journeys as stochastic weighted games exploiting the underlying distribution of events in the logs; (2) a tool chain combining automata learning and model-checking techniques to automatically analyze stochastic user journey games; (3) a method for property-preserving model reduction to visualize the stochastic games results; and (4) the automated stochastic modeling and analysis of two case studies to showcase the usefulness and applicability of the proposed combination of techniques and their extensions.

2 Preliminaries

In the following, we write $\mathcal{D}(X)$ for the set of probability distributions over a set X, where a distribution $\mu\colon X \to [0,1]$ is such that $\sum_{x\in X} \mu(x) = 1$.

Event Logs. An event log records so-called touchpoints (or events) between users and a service provider. A *trace* $\tau = (a_0, \ldots, a_n) \in \mathscr{A}^*$ is a finite, ordered sequence over an alphabet $\mathscr{A}$ of events. An *event log* L is a multi-set of such traces [1]. A *multi-actor event log* $\mathcal{L} = \langle L, \Pi, \alpha \rangle$ assigns an initiating actor to each event in an event log L [26]; the set Π contains a set of actors, and the actor-mapping function $\alpha\colon \mathscr{A} \to \Pi$ assigns events $a \in \mathscr{A}$ to an actor $\pi \in \Pi$.

Automata Learning. To learn stochastic automata from event logs, we use the passive automata learning algorithm IOAlergia [36]. IOAlergia learns stochastic automata for reactive systems defined by MDPs [36], based on Alergia [10]. State merging exploits the underlying probabilities of events in the log. An MDP is a tuple $\langle \Gamma, A_{\text{in}}, A_{\text{out}}, \delta, s_0, \lambda \rangle$ with finite sets of states Γ, input actions A_{in} and output actions A_{out}, a stochastic transition function $\delta\colon \Gamma \times A_{\text{in}} \to \mathcal{D}(\Gamma)$, an initial state $s_0 \in \Gamma$, and a labeling function $\lambda\colon \Gamma \to A_{\text{out}}$. We let $E_\delta \subseteq \Gamma \times A_{\text{in}} \times \Gamma$ denote the finite set of transitions such that $\delta(s,a)(s') > 0$ for all triples $(s,a,s') \in E_\delta$. We assume MDPs to be deterministic; i.e., $s' = s''$ holds for all transitions $\delta(s,a)(s'), \delta(s,a)(s'')$ such that $\delta(s,a)(s') > 0$, $\delta(s,a)(s'') > 0$ and $\lambda(s') = \lambda(s'')$.

Let an input/output log L_{io} consist of traces $\tau_{\text{io}} = (\lambda(s_0), (i_0, o_0), \ldots, (i_n, o_n))$ in which input and output actions alternate, starting with an initial output $\lambda(s_0)$, which is only observed in the initial state. Given L_{io}, IOAlergia creates an input/output frequency prefix tree acceptor (IOFPTA), where states are labeled with output actions and transitions with input actions and frequencies. In the IOFPTA, every path in the tree represents a prefix of a trace in $\tau_{\text{io}} \in L_{\text{io}}$, and the frequency denotes the number of traces sharing this path. After creating the IOFPTA, IOAlergia merges states. Two states are merged if they (1) have the same output label, (2) are locally compatible, and (3) all their successor states with the same output labels are compatible. Local compatibility is based on the Hoeffding bound [25]: two states s, s' are compatible if, for all inputs $i \in A_{\text{in}}$,

$$\left| \frac{f(s,i,o)}{n(s,i)} - \frac{f(s',i,o)}{n(s',i)} \right| \le \sqrt{\frac{1}{2} \log \frac{2}{\epsilon}} \left(\frac{1}{\sqrt{n(s,i)}} + \frac{1}{\sqrt{n(s',i)}} \right),$$

where $f(s,i,o)$ is the frequency of the transition to state o and $n(s,i)$ the sum of frequencies, for input i in state s. The parameter $\epsilon \in (0,2]$ steers the algorithm's eagerness for state merging; e.g., $\epsilon = 2$ leads to no state merges. Therefore, the

MDP might contain several states representing the same event. When no states can be merged, the transition frequencies are normalized to create an MDP.

User Journey Games. A *user journey game* [30,31] is a weighted two-player game $\langle \Gamma, A_C, A_U, E, s_0, T, T_s, w\rangle$, where Γ is a finite set of states, A_C and A_U are disjoint sets of actions, $E \subseteq \Gamma \times A_c \cup A_U \times \Gamma$ is a transition relation, $s_0 \in \Gamma$ an initial state, $T \subseteq \Gamma$ a set of final states, $T_s \subseteq T$ successful final states, and $w : E \to \mathbb{R}$ a weight function. Actions are separated into two disjoint sets: *controllable actions* A_C are taken by the service provider and *uncontrollable actions* A_U by the user. User journey games are *deterministic* if $s' = s''$ for $(s, a, s'), (s, a, s'') \in E$. Uncontrollable actions have higher precedence than controllable actions: hence, the user chooses actions first but might do nothing.

A *stochastic multi-player game* (SMG) [11] is a tuple $\langle \Pi, \Gamma, A, (\Gamma_i)_{i\in\Pi}, s_0, \delta\rangle$, where Π is a set of players, Γ a set of states, A a finite set of actions, $(\Gamma_i)_{i\in\Pi}$ a partition of states among players, $s_0 \in \Gamma$ an initial state, and $\delta : \Gamma \times A \to \mathcal{D}(\Gamma)$ a stochastic transition function. SMGs partition the states among the players; players can take enabled actions if the current state is in their partition. An action $a \in A$ is *enabled* in a state s if there is a transition to another state with non-zero probability, i.e., $\exists s' \in \Gamma : \delta(s, a)(s') > 0$. The set of transitions E_δ defined by δ includes all triples $(s, a, s') \in \Gamma \times A \times \Gamma$ with $\delta(s, a)(s') > 0$. Games can include a reward structure $r : E_\delta \to \mathbb{Q}_{\geq 0}$ mapping transitions to positive rewards (modeling weighted transitions). Rewards accumulate during the game.

Analyzing Stochastic Multiplayer Games. We are interested in analyzing a player's *strategy*, which determines the player's actions in each state. For simplicity, we focus on *memory-less* strategies, where the choice of action is determined by the current state. A *strategy* [11] for player $i \in \Pi$ in an SMG is a partial function $\Gamma_i \to \mathcal{D}(A)$ that maps states to distributions over actions.

PRISM-games [11,32] extends the probabilistic model checker PRISM [34] to games. While PRISM can resolve non-determinism to establish strategies for a single player, PRISM-games can resolve nondeterminism for multiple, possibly competing players. The logic *Probabilistic Alternating-time Temporal Logic with Rewards* (rPATL) allows reasoning about SMGs by expressing temporal properties [11]. The syntax of rPATL is given by:

$$\phi := \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \langle\langle \Xi \rangle\rangle P_{\bowtie q}[\psi] \mid \langle\langle \Xi \rangle\rangle R^r_{\bowtie \chi}[\mathbf{F}^*\, \phi] \mid \langle\langle \Xi \rangle\rangle P_{\blacktriangleright\!\blacktriangleleft}[\psi] \mid \langle\langle \Xi \rangle\rangle R^r_{\blacktriangleright\!\blacktriangleleft}[\mathbf{F}^*\, \phi]$$
$$\psi := \mathbf{X}\phi \mid \phi\mathbf{U}^{\leq k}\phi \mid \phi\mathbf{U}\phi$$

rPATL is a CTL-style branching-time temporal logic that extends state properties ϕ to path formula ψ with probabilistic and reward constraints. Here, p is an atomic proposition. The *coalition* operator $\langle\langle \Xi \rangle\rangle$ denotes the subset $\Xi \subseteq \Pi$ of players that collaborate in a query; these players share a common goal against the remaining adversarial players. The *probabilistic* operator $P_{\bowtie q}$, where $\bowtie \in \{<, \leq, \geq, >\}$ is a comparison operator and $q \in \mathbb{Q} \cap [0, 1]$ is a probability bound, indicates a probabilistic query under bound $\bowtie q$. The *expected cumulative reward* operator $R^r_{\bowtie \chi}$ evaluates the reward structure r for eventually reaching ϕ under bound $\bowtie \chi$, where $\chi \in \mathbb{Q}_{\geq 0}$ is a reward bound and r is a reward structure.

The *quantitative operators* $P_{\bowtie}$ and $R^{r}_{\bowtie}$, with $\bowtie \in \{\min=?, \max=?\}$, return the smallest, respectively largest, value that the given coalition of players Ξ can enforce. The superscript $*$ of the *eventually* operator $\mathbf{F}$ expresses the cost for paths when ϕ is not reached, it may be infinity (∞), zero ($\mathbf{0}$), or accumulated along the path ($\mathbf{c}$). Further temporal logic operators can be constructed from the next operator $\mathbf{X}$, the until operator $\mathbf{U}$, and the bounded until operator $\mathbf{U}^{\leq k}$; for example, the *globally* operator $\mathbf{G}\phi$ is defined via $\mathbf{U}$: $\neg(\top \mathbf{U} \neg\phi)$ [11].

3 Case Study Overview

We conduct two complementary case studies: an industrial application (*GrepS*) and a research benchmark (*BPIC'17*). We explain the steps of our method on GrepS. BPIC'17 includes thousands of journeys and demonstrates scalability.

GrepS. The company GrepS offers programming skill evaluations for Java [6]. The customers of GrepS are organizations that use the service in the hiring process to identify proficient applicants. *Users* of the service, the assessed trainees, usually complete the assessment within 1–2 weeks. The service comprises three phases: (1) sign up, (2) solve all programming tasks, and (3) review and share the skill report with the customer. In a *successful* journey, the user completes all tasks and shares the results with the organization. Otherwise, the journey is *unsuccessful*. The event log contains *anonymized user logs* as tabular data [29]. To construct multi-actor event logs, the actor-mapping function α was detailed by combining domain knowledge and interaction with a GrepS developer.

BPIC'17. The BPI Challenge 2017 captures a loan application process from a bank. Users can *cancel*, *submit* or *complete* applications, and accept phone *calls* from the bank. The process can have three different outcomes: (1) an offer can be accepted by the user, (2) the application can be declined by the bank, or (3) the application can be canceled by the user. We exclude declined applications as they occur due to external factors, e.g., indebtedness. Thus, user journeys are successful if the user accepts one of the provided loan offers; cancellations are unsuccessful. The event log contains anonymized user logs as tabular data [15]. To construct multi-actor event logs, the actor-mapping function α was detailed by combining domain knowledge with information given in the BPIC'17 forum.[1]

Interestingly, BPIC'17 contains a substantial change in the service provider's underlying process, a *concept drift* [2]. To investigate the impact of the concept drift on the user journey, we split the log: The first part (*BPIC'17-1*) contains traces until the change occurred in July 2016, and the second part (*BPIC'17-2*) contains the traces after the change.

The BPIC'17 event log is preprocessed to clear inconsistencies [26, 40]. Specifically, we discretized call durations: A trace might contain several events associated with one call, and calls ranging from seconds to hours. Thus, we aggregate repeated calls and classify them by their duration into "short", "long", or "super long". We exclude calls with an aggregated speaking time of less than 60 seconds.

[1] https://www.win.tue.nl/promforum/categories/-bpi-challenge-2017.

We also distinguish different offers within the same trace. The service provider cancels offers if there is no response after 20 days. We distinguish actively canceled offers and cancellations by the service provider due to timeout. We also found some redundant events; e.g., the event *W_Call after offers* was always followed by *A_Complete*, so we merged these events. To remove outliers we kept only traces that appear more than once in the log; in the end, both logs still contain more than 5000 journeys.

4 From Logs to Stochastic Games

We explain how stochastic user journey games are constructed from multi-actor event logs $\mathcal{L} = \langle L, \Pi, \alpha \rangle$, i.e., the first two steps in Fig. 1. Step 1 generates an MDP M from the multi-actor event log $\mathcal{L}$. Step 2 constructs a weighted stochastic game, extending M with weights and actor information. These *stochastic user journey games* combine user journey games and SMGs (see Sect. 2).

In a multi-actor event log $\mathcal{L}$, the set of actors Π is assumed to include the *service provider* C, who initiates all actions controlled by the offering company, and the *user* U, who initiates all remaining actions. We assume that users engage in only one action at a time; hence, our focus here will be on turn-based games as models for user journeys, and not on models with parallelism.

Step 1. We first learn an MDP $M = \langle \Gamma, A_{\text{in}}, A_{\text{out}}, \delta, s_0, \lambda \rangle$ with IOAlergia. For the construction of M, we make sure that the traces $\tau \in L$ are in the required format of input/output pairs by extending each trace $\tau = (a_0, \ldots, a_n)$ to an input/output trace $\tau_{\text{IO}} = (\lambda(s_0),$ $(\mathsf{env}, \lambda(s_0)^{\alpha(a_0)}), (act(a_0), a_0), \ldots, (\mathsf{env}, a_{n-1}^{\alpha(a_n)}), (act(a_n), a_n), (\mathsf{env}, a_n^{\alpha(res)}),$ $(act(res), res))$. Each $a_i \in \tau$ is encoded by a pair $(\mathsf{env}, a_{i-1}^{\alpha(a_i)})$ where env is a generic input action indicating the next player, followed by an output action $a_{i-1}^{\alpha(a_i)}$ that indicates the player who initiates event a_i from a_{i-1} according to the actor-mapping function α. This pair is followed by a pair $(act(a_i), a_i)$, which uses a function $act\colon \mathscr{A} \to A_{\text{in}}$ to map events to input actions, where the output action corresponds to the event itself. A naive mapping could be $act(a_i) = a_i$, relating each event to a deterministic action. However, it is often useful to introduce a mapping that abstracts slightly from the events to better reflect the problem domain in the actions. Each τ_{IO} starts with an initial output $\lambda(s_0)$ and ends with a final output res, which is $\mathsf{successful}$ if τ records a successful user journey and $\mathsf{unsuccessful}$ otherwise. This resulting set of input/output traces is given to IOAlergia (see Sect. 2). By including input/output pairs $(\mathsf{env}, a_{i-1}^{\alpha(a_i)})$ in the traces, the learned MDP provides the probability distribution for the actions of the next player.

Step 2. The MDP M obtained in Step 1 is extended to a stochastic user journey game by means of a weight function $w : E_\delta \to \mathbb{R}$, labeling transitions with weights, and partitioning the states Γ into service provider states Γ_C and user states Γ_U. For the automatic construction of the weight function w, we exploit the distinction between successful and unsuccessful user journeys in the event log to compute a numerical value that represents the impact of an action on the outcome of the user journey. The calculation of w is based on previous work [30, 31]. For every transition $e \in E_\delta$, we let $w(e) = (1-H(e,L)) \cdot \mathit{majority}(e,L)$, where H is the entropy of successful and unsuccessful journeys. The weight is positive if the majority of traversals are successful journeys, otherwise negative. The weight is maximal, respectively minimal, for transitions occurring exclusively in successful, respectively unsuccessful, journeys. The accumulated weight along a path in a user journey game, called *gas*, then represents the user's "motivation" to continue the journey [30, 31].

Table 1. Model checking queries for SUJGs.

Name	Query	Description
Q1	$\langle\langle C\rangle\rangle P_{\max=?}[\mathbf{F}\,\mathsf{successful}]$	Probability of a successful journey
Q2	$\langle\langle C,U\rangle\rangle R^{\text{NEG}}_{\min=?}[\mathbf{F}\,\mathsf{successful} \mid \mathsf{unsuccessful}]$	Boundaries for accumulated positive and negative rewards
Q3	$\langle\langle C\rangle\rangle R^{\text{NEG}}_{\min=?}[\mathbf{F}\,\mathsf{successful} \mid \mathsf{unsuccessful}]$	
Q4	$\langle\langle C\rangle\rangle R^{\text{POS}}_{\max=?}[\mathbf{F}\,\mathsf{successful} \mid \mathsf{unsuccessful}]$	
Q5	$\langle\langle C\rangle\rangle R^{\text{NEG}}_{\min=?}[\mathbf{C}_{\leq S}]$	Step bounded reward
Q6	$\langle\langle C\rangle\rangle R^{\text{POS}}_{\max=?}[\mathbf{C}_{\leq S}]$	
Q7	$\langle\langle C\rangle\rangle R^{r}_{\max=?}[\mathbf{F}^{\mathbf{c}}\,\mathsf{successful}]$ $r \in \{\text{NEG}, \text{POS}, \text{STEPS}\}$	Expectation of reward structures
Q8	$\langle\langle C\rangle\rangle P_{\max=?}[(\mathbf{F}\,\mathsf{successful}\ \&\ \mathrm{gas} \geq G_0\ \&\ \mathrm{steps} \leq S)\ \&\ (\mathbf{G}\,\mathrm{gas} \geq G_1)]$	Constrained success probability

The controllable and uncontrollable states are identified using the actor-mapping function α to map states to the actors C (service provider) and U (user); e.g., the set of states in Γ_C corresponds to the copies of output actions where C controls the next action: $a_{i-1}^{\alpha(a_i)}$, where $\alpha(a_i) = C$. Then $\Gamma_C = \{s \in \Gamma \mid \exists a \in A_{\text{out}} : \lambda(s) = a^C\}$, and $\Gamma_U = \{s \in \Gamma \mid \exists a \in A_{\text{out}} : \lambda(s) = a^U \vee \lambda(s) = a\}$.

The weight function w and the state partitioning allows the MDP to be transformed into a weighted, two-player SMG, hereafter called a *stochastic user journey game (SUJG)*, i.e., a tuple $G = \langle \{C,U\}, \Gamma, A_{\text{in}}, (\Gamma_i)_{i\in\{C,U\}}, s_0, \delta, T, T_s, w\rangle$, where final states $T = \{s \in \Gamma \mid \lambda(s) = \mathsf{successful} \vee \lambda(s) = \mathsf{unsuccessful}\}$, successful final states $T_s = \{s \in \Gamma \mid \lambda(s) = \mathsf{successful}\}$, and w the weight function. Note that every user journey game can be transformed into an equivalent SUJG.

5 Queries for Stochastic User Journey Games

We here assume that users do not interact infinitely with a service provider but eventually stop. Therefore, we consider SUJGs to be *stopping* games, in which we reach almost surely terminal states with reward zero [33].

Step 3. We now consider the probabilistic model checking of properties that are crucial for the success of user journeys. The violation of these properties allows us to locate problematic states where the user journey may be improved. The constructed SUJG may contain loops with a positive or negative sum of weights. For this reason, we distinguish queries applicable to games with *reward structures* and with *bounded integer encodings*. Table 1 lists properties that we analyzed for the case studies, and that we discuss below. The queries are specified in rPATL, where C denotes the service provider and U denotes the user.

Let us first analyze the probability of completing a user journey successfully; i.e., to what extent can service provider C guarantee the successful outcome of the game? Query Q1 quantifies the service provider's ability to guide an independent user. Searching for states that return a small probability of reaching any $s \in T_s$ uncovers states from which the service provider has little or no probability of successfully guiding the user. Thus, the journey is likely to fail. Here, $\mathsf{successful}$ is a predicate that only holds in the successful final state T_s, and $\mathsf{unsuccessful}$ is a predicate that holds in the final states $T \setminus T_s$.

Reward Structures decouple accumulated rewards from the state space in PRISM-games and allow efficient computation of accumulated rewards. In turn-based SMGs, PRISM-games only supports positive rewards. Thus, we use two reward structures: POS for positive and NEG for negative gas (see Sect. 4). The weight of a transition in the SUJG contributes to the corresponding structure, i.e., positive weights add to POS, and negative weights add to NEG. Many services contain transitions with negative weights, e.g., reflecting actions that may be unintuitive for the user. To analyze the effect of these transitions, we consider queries concerning the user experience. Query Q2 determines the lower bound for the negative reward that the user must accumulate to achieve any outcome, by assuming that both actors cooperate. Queries Q3 and Q4 determine the minimum NEG and maximum POS reward that the service provider can guarantee, independent of the user, over successful and unsuccessful journeys, respectively. Rewards can also be used to relate gas to the number of steps taken so far: Queries Q5 and Q6 return the minimum negative or maximum positive accumulated reward (denoted $\mathbf{C}$) within the first S steps that C can guarantee.

Bounded Integer Encodings combine positive and negative weights in one variable, enabling queries on their difference. Every transition changes the value of this variable by the corresponding positive or negative weight, reflecting the gas along the paths in the game (see Sect. 4). We also consider a step counter that is updated for each transition. To restrict the size of the search space, we give this variable a bound (i.e., steps $:= \min(\text{steps}+1, X)$ for some X). We then use concentration inequalities such as *Markov's inequality* and cumulative reward structures to calculate the expected values of POS, NEG, and STEPS in Q7, and derive upper and lower bounds that include at least a minimum part of the distribution. Note that this construction is only needed in the presence of loops and that the *expected total rewards*, used to bound the model, are finite as we assume stopping games. Query Q8 determines the service provider's probability for a successful journey with a minimum amount of gas along the path, a maximum amount of steps, and an overall lower bound for the gas. This multi-objective query searches for a successful final state where gas $\geq G_0$ and steps $\leq S$, while ensuring that gas never decreases below G_1, for constants G_0, S, G_1.

Experiments. PRISM-games supports *experiments* on queries that instantiate a variable, e.g., the maximum number of steps, with all values in a given integer interval. We use experiments to compare different values of player activity by modifying the probabilities for the service provider or user to take their actions first. Additionally, we vary the allowed number of steps to investigate how the probabilities of a successful outcome change with a limited number of steps.

6 Model Reduction for Visualization

Model checking may reveal weaknesses in the service design and unsatisfiable queries may suggest a need for changes. However, an unsatisfiable query does not by itself identify the actions that negatively affect the largest number of users. To help prioritize options during service redesign, we rank actions based on their expected influence on the user journey outcome, to identify the most critical actions for the largest number of users (cf. Step 3, Fig. 1). We synthesize strategies maximizing the probability of a successful outcome by returning a maximizing strategy for the service provider and a minimizing strategy for the user, based on the queries in Sect. 5. These strategies resolve the players' choice of action in the SUJG via an induced Markov chain $M' = \langle \Gamma', \delta', s_0 \rangle$; the states Γ' of M' form a, possibly smaller, subset of the states Γ of the original SUJG, i.e., $\Gamma' \subseteq \Gamma$. (The construction of the induced Markov chain M' from an SMG is detailed in [12].)

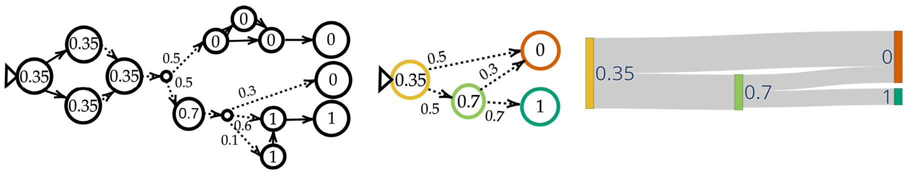
We say that users are *guidable* if the probability that they can successfully complete the journey is greater than zero. Let the function $\mathscr{R} : \Gamma' \rightarrow [0, 1]$ map states $s \in \Gamma'$ to the (intermediate) results of the probabilistic query Q1, expressing the probability of reaching the successful outcome from s. The difference in guidable users between two neighboring states s and s' is the absolute difference between $\mathscr{R}(s)$ and $\mathscr{R}(s')$, multiplied by the users traversing between these states. Formally, the *difference* diff: $\Gamma' \rightarrow \mathbb{R}$ in state $s \in \Gamma'$ is the absolute difference in guidable users between s and all neighboring states s':

$$\text{diff}(s) = \sum_{s' \in \Gamma'} |\mathscr{R}(s) - \mathscr{R}(s')| \cdot \#_{\mathcal{L}}^{\Gamma'}(s, s') \ . \tag{1}$$

Here, $\#_{\mathcal{L}}^{\Gamma'}(s, s')$ denotes the number of users traversing from s to s' as recorded in the log $\mathcal{L}$, where $s' \in \Gamma'$ and $\delta'(s, s') > 0$. For non-neighboring states, let $\#_{\mathcal{L}}^{\Gamma'}(s, s') = 0$. States can then be ranked in descending order by their difference.

Visualizations of Results. Real-world processes with complex structures and many users result in models that might be hard for humans to interpret correctly. We discuss a model visualization method based on the model-checking results that allows model reduction while preserving the ranking order.

The state space of M' can be abstracted into clusters of states with an equal probability of success as defined by $\mathscr{R}$. Neighboring states with the same results can be merged. States $\{s' \in \Gamma' \mid (s, s') \in E_{\delta'} \wedge \mathscr{R}(s) = \mathscr{R}(s')\}$ can be merged into



(a) SUJG annotated with model checking results.

(b) Reduced Markov chain.

(c) Sankey diagram generated from the reduced Markov chain.

Fig. 2. We visualize the model checking results in a Sankey diagram that is generated from the learned (SUJGs).

a state s. We also merge successful final states $T_s \cap \Gamma'$ and unsuccessful final states $(T \setminus T_s) \cap \Gamma'$. Note that the reduced model preserves all transitions to states that negatively impact the user journey, and that the merge operation is commutative.

To visualize fluctuations in guidable users along the user journey, we transform the reduced model into a *Sankey* diagram [39]. We opted for Sankey diagrams since they seem accessible to a wide range of stakeholders with some previous insights into the user behavior [19]. Each bar in the diagram illustrates changes in guidable users, divided into flows of lost and gained guidable users. The largest bars indicate states that are promising candidates for improvement. Note that the bars are not monotonic as they do not visualize the absolute number of users in a state, but the weighted difference in guidable users.

A heat map visualizes the result mapping $\mathscr{R}$ in the reduced Markov chain. By clustering similar states, we can keep diagrams fairly small without compromising the analysis. Figure 2a shows a SUJG with three necessary user actions to reach a successful outcome. States are annotated with the probability of reaching the successful final state, dotted lines represent uncontrollable user actions, annotated with their probabilities. Figure 2b shows the *reduced Markov chain*, where two actions divide the states into four clusters with 35%, 70%, 100%, and 0% probability of success, respectively. The insights gained from the induced Markov chain are then visualized as a Sankey diagram in Fig. 2c. The example illustrates flow capacities through the distribution of 100 users.

7 Case Study Results

We present results for the GrepS and BPIC'17 case studies from Sect. 3. The steps described in Sects. 4–6 are assembled in a tool chain, implemented in Python 3.10.12, and available online [27]. For automata learning, we use the IOAlergia implementation of AALpy [37] (v. 1.4) and, for model checking, PRISM-games [11,32] (v. 3.2.1). All experiments ran on a laptop with 32 GB memory and an i7-1165G7 @ 2.8 GHz Intel processor within few hours.

Table 2. Model checking results for GrepS and BPIC'17.

Name	GrepS	BPIC'17-1	BPIC'17-2
Q2	16.49	33.11	33.87
Q3	50.55	37.35	36.07
Q4	44.98	67.79	68.07

GrepS. Figure 3 shows the generated cyclic game, where touchpoints are represented as states, identified by T and a number. It encodes a heat map, ranging from yellow states to green states; the darker a state's green, the greater its probability for success (orange is the unsuccessful state). Transitions with negative weights are orange, and those with positive weights green. The figure highlights the three phases of GrepS' user journey. Phase 1 consists of touchpoints T0–T4, Phase 2 of T5–T20 and Phase 3 of T21–T26. Users receive a new task in T9, T11, T13, T15, and T17. Feedback to users is given after every task. Users share their results with the client company in T26. For readability, we merged the service-provider controlled and user controlled states, which we introduced due to the input/output format of the traces, see Step 1 in Sect. 4, with their preceding touchpoint-labeled states (the full model is available at [27]). For GrepS, we assume that users, when it is their turn, can transition according to the recorded events, or do nothing, i.e., transition to a service-provider state, if available.

We investigate the limits for the positive and negative weights that the service provider can guarantee during the journey, with the user and on its own. Table 2

presents results for model checking the queries Q2–Q4 (see Table 1) for both case studies. For GrepS, the user must endure a significant number of negatively weighted transitions, since the maximum accumulated POS (Q4) is smaller than the minimum accumulated NEG (Q3). Cooperation (Q2) results in a 67.37% reduction in accumulated NEG.

We analyze the impact of the users' and service provider's activity on the user journey by varying the probability in the game's transitions, to change how eager a player is in taking action. Figure 4a shows the results for these changes: on the horizontal axis, $q = 0$ means that the player takes action according to the frequencies of the original game, $-1 \leq q < 0$ means that the service provider gradually increases the probability of taking action (the service provider always takes an action, if available, with $q = -1$). Similarly, for $1 \geq q > 0$, the user gradually increases the probability of taking action (until always taking an action, if available, with $q = 1$). The vertical axis shows the

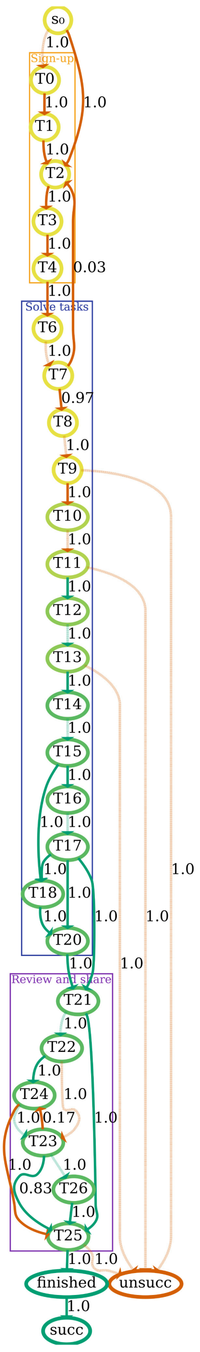
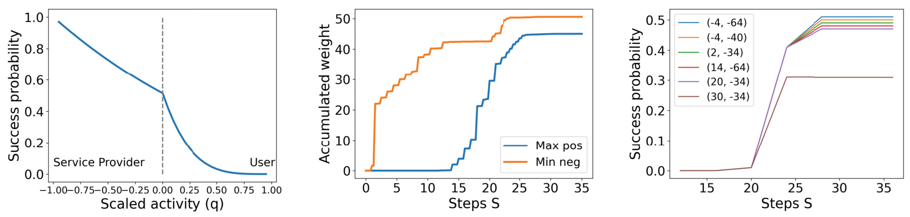


Fig. 3. Simplified model of GrepS' user journey.



(a) Parametric eagerness of the players (Query Q1)

(b) Gas by steps (Queries Q5 & Q6)

(c) Bounded experiment (Query Q8) over (G_0, G_1)

Fig. 4. Experiment results for the GrepS case study.

probability of a successful journey (Q1); interestingly, GrepS has a linear gain from being more active and a non-linear loss from being more passive. Figure 4b shows the results for queries Q5 and Q6 by comparing the maximal accumulated positive and the minimum accumulated negative weights for the first S steps of the journey, revealing that negative weights surpass positive weights, especially at the beginning of a journey.

To evaluate whether the service provider can guide users to a successful outcome with limited steps and lower bounds for the gas, we consider the model with bounds derived from query Q7 (see Sect. 5). We bound the integer encodings by 10 times their expected value, which includes at least 90% of the traces. Figure 4c shows the development in guiding the user under Q8. The plot's labels are pairs (G_0, G_1), where G_0 is the minimum gas in the final state and G_1 the lower bound for gas along the journey. For pairs with the same results, we only plot pairs with the maximum final gas and the maximum gas along the journey. The plot shows that experiencing a journey with high minimal gas and reaching a successful outcome are conflicting goals; maximizing minimal gas clearly affects the probability of success for the user journey. For the best probability of success (51%), GrepS needs to guide the users through the negatively weighted transitions, which reach a minimum gas of -64. Actually, the user never fully recovers positive gas in this journey, which ends with a negative gas of -4.

The analysis has shown that users face negative experiences and that the service provider can offer guidance. We now consider where the journey can be improved to help users reach a successful outcome. Figure 5 shows the derived Sankey diagram with observed users as flow capacities, as described in Sect. 6. The reduced model contains only 6 states, while the mined one has 65 states. Based on the state ranking function (Eq. 1), state T25, where users accept or reject their test results, appears as the most critical state for a successful journey; it determines whether the user will (or not) reach a successful final state; in fact, 25% percent of the users recorded in the log fail their journey immediately after this state. The second most critical state is the first task T9 (where 37.5% of all users are lost), followed by the other tasks. However, at these points in the journey, several user-controlled actions are required for a successful journey, which makes GrepS dependent on the user's cooperation in these states.

Thus, the SUJGs allow us to identify specific states for enhancing the journey: T9 and T25. Our analysis clearly shows that GrepS needs to be active to achieve a successful user journey (Fig. 4). We note that most negatively weighted transitions are user-controlled, suggesting that GrepS can prevent users from "derailing" from a successful journey by being more active within the user journey. If GrepS provides less guidance, users tend to abandon their journeys more easily.

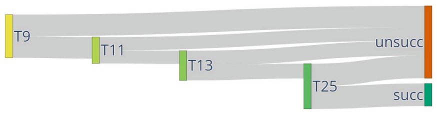


Fig. 5. Sankey diagram of Greps' user journey for guidable users.

Stakeholder validation of GrepS Results. We presented the results obtained for GrepS to a company stakeholder[2] to obtain feedback on our results and their presentation format. The stakeholder was not involved in performing the case study; the other authors only had access to the event log from GrepS, provided in 2021. This validation was done after the analysis results were available.

He was familiar with Sankey diagrams and immediately observed that our analysis makes non-trivial insights accessible to key-stakeholders, varying from concrete recommendations to non-trivial prescriptions on company behavior. From the company's perspective, prioritizing limited resources to improve the users' success rate and experience is challenging. Our case study substantiates that automated analyses based on event logs are a viable alternative to current best-practices based on heuristics, and promise to reduce assessment efforts.

The identification of T25 as a candidate for improvement (Fig. 5) had actually been discovered independently by GrepS, confirming our analysis. This step is currently supplemented by a manual follow-up step, since completing the user journey successfully is crucial to provide a good user experience. The second suggested task, T9, is not obvious to GrepS and introduces options they have not yet considered, namely to spend resources on guiding the user rather than further optimizing the negative weighted sign-up phase (see Fig. 4b).

The analysis of actor eagerness related to the probability of success (Fig. 4a) is novel and implies that revenue from resources invested in guiding users can be computed. This allows GrepS to evaluate whether to spend more resources on guiding users, given the linear scaling of success probability, or to cut costs through less guidance, reducing manual work while increasing service adversity.

Figures 4b and 4c can be used to relate user profiles and user journeys. A user's motivation to complete tests and share results despite negatively weighted actions, is initially unknown. If the company had some prior knowledge about the initial motivation of a user or a group of users, it would be possible to model different journeys through the service. In particular, Fig. 4c can support such endeavors, because different bounds can be identified for different planned journeys with corresponding probabilities for success.

[2] The third author of this paper is a long-term stakeholder of GrepS.

BPIC'17. Applying Steps 1 and 2 to BPIC'17 yields models with 95 states for BPIC'17-1 and 131 for BPIC'17-2. Step 3 reduces the models to 32 and 47 states, respectively (i.e., +60% reduction). When filtering on reachable states, using the generated strategy, the models shrink to 15 and 19 states, respectively. Figure 7 shows the Sankey diagrams for the two event logs. For readability, we omit the names of states with the least difference in guidable users and use a heat map as in Fig. 3.

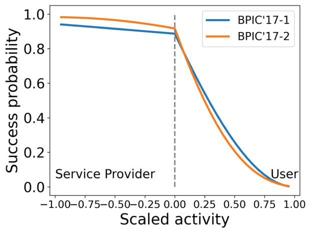
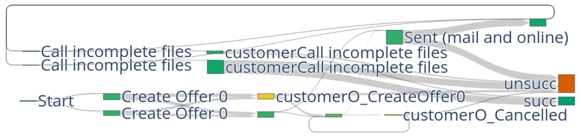


Fig. 6. Parametric eagerness for Q1 in BPIC'17.

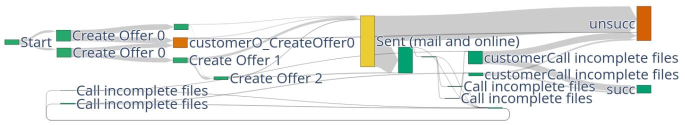
The comparison of model checking results between the two models with queries Q2–Q4 (see Table 2) shows some small improvements from BPIC'17-1 to BPIC'17-2. Figure 6 compares different levels of player eagerness for both SUJGs, model checking Q1. It reveals improvements in the service. BPIC'17-2 outperforms BPIC'17-1 starting from $q = 0.06$ when increasing the service provider's probability to take an action. (Plots showing results for the remaining queries, similar to the queries for the GrepS case study, are available online [27].)

Figure 7 shows the positive impact for BPIC'17-2 after the concept drift. In BPIC'17-1, the number of guidable users remains constant through the user journey, with the most critical state causing only 27% of the total user difference. In BPIC'17-2, the main critical state causes a total of 50% difference of guidable users. We also observe a change in loan offers: the 2nd and 3rd offers are prominent in the reduced BPIC'17-2 model (while they were merged with other states or omitted in BPIC'17-1), each with decreasing flow capacity. Furthermore, the probability of guiding users from "*customer Create Offer 0*" reduced; this state is marked yellow in BPIC'17-1 and orange in BPIC'17-2, indicating a decrease in user experience. In both journeys, the second most critical state, a short call due to incomplete files, is user-controlled, but its fraction of the total guidable user's difference decreased from 26.6% to 12.5%. This can be interpreted as evidence that the service provider improved this call state after the concept drift. However, we observe that BPIC'17-2 still lacks proper guidance for the effect of the call, based on the direct transition to the unsuccessful final state.

Threats to Validity. For model learning with IOAlergia, we set the parameter ϵ (which regulates state merging) according to the size of the underlying event log and the assumed complexity of the service. For GrepS, we set $\epsilon = 0.1$ due to a small number of possible journeys, while for BPIC'17, we set $\epsilon = 0.8$ to capture different decisions and possible executions. Insights from GrepS highly depend on ϵ, where a larger ϵ restricts state merging. For BPIC'17, we observe that the eagerness experiment (Fig. 6) replicates for various ϵ values, though with variations for either small or large ϵ values. Further investigations are needed to draw rigorous conclusions about this relation. The model-checking analysis in Step 3, which generate Sankey diagrams, do not require a minimal flow of users. Strategies might exploit rarely observed behavior, they do not consider a minimum bound for the coverage of users. Table 1 presented queries that target



(a) BPIC'17-1



(b) BPIC'17-2

Fig. 7. Sankey diagrams generated from the reduced BPIC'17 models.

Pareto optimization problems to optimize multiple conflicting objectives, e.g., limited steps and minimal gas in positive states. We explored solutions to these problems with PRISM-games experiments, but one could also search for all solutions. The efficiency of our technique depends on automata learning and model checking; all presented results are reproducible within $\sim 9\,$h.

8 Related Work

Related work primarily focuses on designing domain-specific modeling languages that allow modeling from the user's perspective. The methods developed [5,9,14,20,22,23,35,38,41] concentrate on manually constructing user journeys based on expert knowledge [9], user questionnaires [21,41], or given event logs [5]. The analysis of the resulting models is typically also performed manually. However, Lammel *et al.* [35] propose an ontology-based technique that allows the automatic generation of visualizations to provide further insights.

Process discovery [1] is a technique to automatically generate models from event logs and has been applied to generate different types of user journey models such as customer journey maps (CJM) [7,8,24] or transition systems [26,28,30,31,42]. CJMs represent grouped traces in the event logging, unlike our work where we mine a general model. Existing approaches [26,28,31,43] that use process discovery techniques to mine transition systems ignore the underlying distribution of events. By capturing the probabilities in the model, we can perform a finer analysis and visualization, and provide guidelines to the service provider in case of changing behavior. In our previous work [31], we also generated weighted deterministic user journey games and applied model checking to find bottlenecks in the service. By applying automata learning instead of process discovery techniques, we enhance this approach to generate probabilistic games.

Automata learning techniques [3,13,16,45] have been used to mine process models, e.g., transition systems or Petri nets, from given event logs. However, our proposed approach incorporates the users' perspective. While existing techniques may also consider the underlying probability distribution of the event log constructing the model, they neglect it for later analysis. Wieman *et al.* [45] derive improvements for industrial case studies manually from the learned model.

9 Conclusion

This paper presents two complementary case studies for the automated modeling and analysis of user journeys from event logs. Our analysis tool chain combines automata learning and model-checking techniques, based on a formalization of user journeys as stochastic weighted games that exploits the underlying distribution of events in the log. Model-checking results are used in property-preserving model reduction, which allows us to automatically identify and rank actions that are critical to the outcome of the user journey and visualize their effect. To the best of our knowledge, this is the first work using stochastic games in an automated method to analyze and improve user journeys.

The investigated case studies demonstrate the applicability of our approach to real-world services, varying in size and complexity. The results of the case studies lead us to three main observations: (1) model visualization creates compact Sankey diagrams for complex services that facilitate the interpretation of formal analyses; (2) the model reduction preserves changes in the underlying journeys, e.g., the concept drift for BPIC'17; and (3) the state ranking method effectively identifies candidate states for service redesign, based on user experience. Compared to previous work, our exploitation of the underlying probabilistic distribution of events enabled a more targeted analysis of the user journeys. For future work, automatically capturing the actor information in the event logs would make our approach less dependent on domain knowledge.

Data Availability Statement. The artifact to replicate the presented results is publicly available on Zenodo at https://doi.org/10.5281/zenodo.12529995.

References

1. van der Aalst, W.M.P.: Process Mining - Data Science in Action. Springer, 2 edn. (2016). https://doi.org/10.1007/978-3-662-49851-4
2. Adams, J.N., Zelst, S.J.v., Quack, L., Hausmann, K., van der Aalst, W.M., Rose, T.: A framework for explainable concept drift detection in process mining. In: International Conference on Business Process Management, vol. 12875, pp. 400–416. Springer (2021). https://doi.org/10.1007/978-3-030-85469-0_25
3. Agostinelli, S., Chiariello, F., Maggi, F.M., Marrella, A., Patrizi, F.: Process mining meets model learning: discovering deterministic finite state automata from event logs for business process analysis. Inf. Syst. **114**, 102180 (2023). https://doi.org/10.1016/J.IS.2023.102180
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)

5. Berendes, C.I., Bartelheimer, C., Betzing, J.H., Beverungen, D.: Data-driven customer journey mapping in local high streets: a domain-specific modeling language. In: Pries-Heje, J., Ram, S., Rosemann, M. (eds.) Proc. International Conference on Information Systems (ICIS 2018). Association for Information Systems (2018). https://aisel.aisnet.org/icis2018/modeling/Presentations/4
6. Bergersen, G.R., Sjøberg, D.I.K., Dybå, T.: Construction and validation of an instrument for measuring programming skill. IEEE Trans. Softw. Eng. **40**(12), 1163–1184 (2014). https://doi.org/10.1109/TSE.2014.2348997
7. Bernard, G., Andritsos, P.: CJM-ab: abstracting customer journey maps using process mining. In: Mendling, J., Mouratidis, H. (eds.) Information Systems in the Big Data Era - Proceedings CAiSE Forum 2018. Lecture Notes in Business Information Processing, vol. 317, pp. 49–56. Springer (2018), https://doi.org/10.1007/978-3-319-92901-9_5
8. Bernard, G., Andritsos, P.: Contextual and behavioral customer journey discovery using a genetic approach. In: Welzer, T., Eder, J., Podgorelec, V., Latific, A.K. (eds.) Proceedings 23rd European Conference on Advances in Databases and Information Systems (ADBIS 2019). Lecture Notes in Computer Science, vol. 11695, pp. 251–266. Springer (2019), https://doi.org/10.1007/978-3-030-28730-6_16
9. Bitner, M.J., Ostrom, A.L., Morgan, F.N.: Service blueprinting: a practical technique for service innovation. Calif. Manage. Rev. **50**(3), 66–94 (2008). https://doi.org/10.2307/41166446
10. Carrasco, R.C., Oncina, J.: Learning stochastic regular grammars by means of a state merging method. In: Carrasco, R.C., Oncina, J. (eds.) Proceedings Second International Colloquium on Grammatical Inference and Applications (ICGI-94), Lecture Notes in Computer Science, vol. 862, pp. 139–152. Springer (1994). https://doi.org/10.1007/3-540-58473-0_144
11. Chen, T., Forejt, V., Kwiatkowska, M.Z., Parker, D., Simaitis, A.: Automatic verification of competitive stochastic systems. Formal Methods Syst. Des.**43**(1), 61–92 (2013). https://doi.org/10.1007/S10703-013-0183-7
12. Chen, T., Forejt, V., Kwiatkowska, M.Z., Simaitis, A., Trivedi, A., Ummels, M.: Playing stochastic games precisely. In: Koutny, M., Ulidowski, I. (eds.) Proceedings 23rd International Conference on Concurrency Theory (CONCUR 2012), Lecture Notes in Computer Science, vol. 7454, pp. 348–363. Springer (2012). https://doi.org/10.1007/978-3-642-32940-1_25
13. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. (TOSEM) **7**(3), 215–249 (1998). https://doi.org/10.1145/287000.287001
14. Crosier, A., Handford, A.: Customer journey mapping as an advocacy tool for disabled people: a case study. Soc. Mark. Quart. **18**(1), 67–76 (2012). https://doi.org/10.1177/1524500411435483
15. van Dongen, B.: BPI Challenge 2017 (2017). https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b
16. Esparza, J., Leucker, M., Schlund, M.: Learning workflow Petri nets. Fundam. Informaticae **113**(3-4), 205–228 (2011). https://doi.org/10.3233/FI-2011-607
17. Fornell, C., Mithas, S., Morgeson, F.V., Krishnan, M.: Customer satisfaction and stock prices: high returns, low risk. J. Mark. **70**(1), 3–14 (2006). https://doi.org/10.1509/jmkg.70.1.003.qxd
18. Gold, E.M.: Language identification in the limit. Inf. Control **10**(5), 447–474 (1967). https://doi.org/10.1016/S0019-9958(67)91165-5

19. Gutwin, C., Mairena, A., Bandi, V.: Showing flow: comparing usability of Chord and Sankey diagrams. In: Schmidt, A., Väänänen, K., Goyal, T., Kristensson, P.O., Peters, A., Mueller, S., Williamson, J.R., Wilson, M.L. (eds.) Proceedings 2023 Conference on Human Factors in Computing Systems (CHI 2023), pp. 825:1–825:10. ACM (2023). https://doi.org/10.1145/3544548.3581119
20. Halvorsrud, R., Boletsis, C., Garcia-Ceja, E.: Designing a modeling language for customer journeys: lessons learned from user involvement. In: Proceedings 24th International Conference on Model Driven Engineering Languages and Systems (MODELS 2021), pp. 239–249. IEEE (2021). https://doi.org/10.1109/MODELS50736.2021.00032
21. Halvorsrud, R., Haugstveit, I.M., Pultier, A.: Evaluation of a modelling language for customer journeys. In: Blackwell, A.F., Plimmer, B., Stapleton, G. (eds.) Proceedings Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2016), pp. 40–48. IEEE Computer Society (2016). https://doi.org/10.1109/VLHCC.2016.7739662
22. Halvorsrud, R., Kvale, K., Følstad, A.: Improving service quality through customer journey analysis. J. Ser. Theor. Pract. **26**(6), 840–867 (2016). https://doi.org/10.1108/JSTP-05-2015-0111
23. Halvorsrud, R., Mannhardt, F., Johnsen, E.B., Tapia Tarifa, S.L.: Smart journey mining for improved service quality. In: Carminati, B., et al. (eds.) Proceedings International Conference on Services Computing (SCC 2021), pp. 367–369. IEEE (2021). https://doi.org/10.1109/SCC53864.2021.00051
24. Harbich, M., Bernard, G., Berkes, P., Garbinato, B., Andritsos, P.: Discovering customer journey maps using a mixture of Markov models. In: Ceravolo, P., van Keulen, M., Stoffel, K. (eds.) Proceedings 7th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2017). CEUR Workshop Proceedings, vol. 2016, pp. 3–7. CEUR-WS.org (2017). http://ceur-ws.org/Vol-2016/paper1.pdf
25. Hoeffding, W.: Probability inequalities for sums of bounded random variables. In: Fisher, N.I., Sen, P.K. (eds.) The Collected Works of Wassily Hoeffding, pp. 409–426. Springer (1994). https://doi.org/10.1007/978-1-4612-0865-5_26
26. Kobialka, P., Mannhardt, F., Tapia Tarifa, S.L., Johnsen, E.B.: Building user journey games from multi-party event logs. In: Proceedings 3rd International Workshop on Event Data and Behavioral Analytics (EdbA 2022), Lecture Notes in Business Information Processing, vol. 468. Springer (2022). https://doi.org/10.1007/978-3-031-27815-0_6
27. Kobialka, P., Pferscher, A., Johnsen, E.B., Tapia Tarifa, S.L.: Supplementary material: stochastic games for user journeys. https://github.com/smartjourneymining/probabilistic_games/releases/tag/FM2024 (2024)
28. Kobialka, P., Schlatte, R., Bergersen, G.R., Johnsen, E.B., Tapia Tarifa, S.L.: Simulating user journeys with active objects. In: de Boer, F.S., Damiani, F., Hähnle, R., Johnsen, E.B., Kamburjan, E. (eds.) Active Object Languages: Current Research Trends, LNCS, vol. 14360, pp. 199–225. Springer (2024). https://doi.org/10.1007/978-3-031-51060-1_8
29. Kobialka, P., Tapia Tarifa, S.L., Bergersen, G.R., Johnsen, E.B.: Weighted games for user journeys (data set). https://doi.org/10.5281/zenodo.6962413 (2022). Accessed 01 April 2024
30. Kobialka, P., Tapia Tarifa, S.L., Bergersen, G.R., Johnsen, E.B.: Weighted games for user journeys. In: Schlingloff, B., Chai, M. (eds.) Proc. 20th International Conference on Software Engineering and Formal Methods (SEFM 2022), LNCS, vol.

13550, pp. 253–270. Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_16
31. Kobialka, P., Tapia Tarifa, S.L., Bergersen, G.R., Johnsen, E.B.: User journey games: Automating user-centric analysis. Softw. Syst. Model. **23**(3), 605–624 (2024). https://doi.org/10.1007/s10270-024-01148-2
32. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: PRISM-games 3.0: Stochastic game verification with concurrency, equilibria and time. In: Lahiri, S.K., Wang, C. (eds.) Proceedings 32nd International Conference on Computer Aided Verification (CAV 2020), LNCS, vol. 12225, pp. 475–487. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_25
33. Kwiatkowska, M., Parker, D., Wiltsche, C.: PRISM-games: verification and strategy synthesis for stochastic multi-player games with multiple objectives. Int. J. Softw. Tools Technol. Transf. **20**(2), 195–210 (2018). https://doi.org/10.1007/S10009-017-0476-Z
34. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Procedings 23rd International Conference on Computer Aided Verification (CAV 2011), LNCS, vol. 6806, pp. 585–591. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_47
35. Lammel, B., Korkut, S., Hinkelmann, K.: Customer experience modelling and analysis framework - a semantic lifting approach for analyzing customer experience. In: Proceedings 6th International Conference on Innovation and Entrepreneurship (IE 2016). GSTF (Dec 2016). http://hdl.handle.net/11654/24293
36. Mao, H., Chen, Y., Jaeger, M., Nielsen, T.D., Larsen, K.G., Nielsen, B.: Learning deterministic probabilistic automata from a model checking perspective. Mach. Learn.**105**(2), 255–299 (2016). https://doi.org/10.1007/S10994-016-5565-9
37. Muškardin, E., Aichernig, B.K., Pill, I., Pferscher, A., Tappler, M.: AALpy: an active automata learning library. Innovations Syst. Softw. Eng. **18**(3), 417–426 (2022). https://doi.org/10.1007/S11334-022-00449-3
38. Razo-Zapata, I.S., Chew, E.K., Proper, E.: VIVA: a visual language to design value co-creation. In: Proceedings 20th Conference on Business Informatics (CBI 2018), vol. 01, pp. 20–29. IEEE (2018). https://doi.org/10.1109/CBI.2018.00012
39. Riehmann, P., Hanfler, M., Froehlich, B.: Interactive Sankey diagrams. In: Stasko, J.T., Ward, M.O. (eds.) IEEE Symposium on Information Visualization (InfoVis 2005), pp. 233–240. IEEE Computer Society (2005). https://doi.org/10.1109/INFVIS.2005.1532152
40. Rodrigues, A.M.B., et al.: Stairway to value: mining a loan application process (2017). https://www.win.tue.nl/bpi/2017/bpi2017_winner_academic.pdf
41. Rosenbaum, M.S., Otalora, M.L., Ramírez, G.C.: How to create a realistic customer journey map. Bus. Horiz. **60**(1), 143–150 (2017). https://doi.org/10.1016/j.bushor.2016.09.010
42. Terragni, A., Hassani, M.: Analyzing customer journey with process mining: from discovery to recommendations. In: Proceedings 6th International Conference on Future Internet of Things and Cloud (FiCloud 2018), pp. 224–229. IEEE (Aug 2018). https://doi.org/10.1109/FiCloud.2018.00040
43. Terragni, A., Hassani, M.: Optimizing customer journey using process mining and sequence-aware recommendation. In: Proceedings 34th Symposium on Applied Computing (SAC 2019), pp. 57–65. ACM Press (Apr 2019). https://doi.org/10.1145/3297280.3297288

44. Vandermerwe, S., Rada, J.: Servitization of business: Adding value by adding services. Eur. Manage. J. **6**(4), 314–324 (1988). https://doi.org/10.1016/0263-2373(88)90033-3
45. Wieman, R., Aniche, M.F., Lobbezoo, W., Verwer, S., van Deursen, A.: An experience report on applying passive learning in a large-scale payment company. In: Proceeedings International Conference on Software Maintenance and Evolution (ICSME 2017), pp. 564–573. IEEE Computer Society (2017).https://doi.org/10.1109/ICSME.2017.71

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Embedded Systems Track

Compositional Verification of Cryptographic Circuits Against Fault Injection Attacks

Huiyu Tan[1,2], Xi Yang[1], Fu Song[3,4](✉), Taolue Chen[5], and Zhilin Wu[3]

[1] ShanghaiTech University, Shanghai 201210, China
[2] Wingsemi Technology Co., Ltd., Shanghai 201203, China
[3] Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
{wuzl,songfu}@ios.ac.cn
[4] Nanjing Institute of Software Technology, Nanjing 211135, China
[5] Birkbeck, University of London, London WC1E 7HX, UK
t.chen@bbk.ac.uk

Abstract. Fault injection attack is a class of active, physical attacks against cryptographic circuits. The design and implementation of countermeasures against such attacks are intricate, error-prone and laborious, necessitating formal verification to guarantee their correctness. In this paper, we propose the first compositional verification approach for round-based hardware implementations of cryptographic algorithms. Our approach decomposes a circuit into a set of single-round sub-circuits which are verified individually by either SAT/SMT- or BDD-based tools. Our approach is implemented as an open-source tool CLEAVE, which is evaluated extensively on realistic cryptographic circuit benchmarks. The experimental results show that our approach is significantly more effective and efficient than the state-of-the-art.

1 Introduction

Cryptographic circuits are widely applied in various embedded and cyber-physical systems [5,39]. However, they are vulnerable to fault injection attacks, which disrupt the execution of cryptographic primitives via clock glitch [2], underpowering [34], voltage glitch [41], electromagnetic pulse [16], or laser beam [36]. With circuit's faulty outputs, attackers can employ statistical analysis methods to infer sensitive information, thereby threatening the security of,

This work was funded by the Strategic Priority Research Program of CAS (XDA0320101), National Natural Science Foundation of China (62072309), CAS Project for Young Scientists in Basic Research (YSBR-040), ISCAS New Cultivation Project (ISCAS-PYFX-202201), ISCAS Fundamental Research Project (ISCAS-JCZD-202302), oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2023A04).

© The Author(s) 2025

e.g., authentication. As a result, fault injection attacks pose a significant threat to the security of embedded and cyber-physical systems.

While countermeasures have been proposed to mitigate these attacks [1,26,35], their implementation does not necessarily guarantee security. Crucially, the fault-resistance of these countermeasures needs to be formally verified. While a plethora of fault-resistance analysis approaches have been proposed (cf. Sect. 6), the state-of-the-art formal verification approaches are non-compositional and limited in efficiency and scalability for realistic cryptographic circuits.

Contributions. In this work, we propose the first compositional verification approach for sequential circuits of cryptographic primitives with countermeasures against fault injection attacks, aiming to combat the efficiency and scalability challenges. Different from existing approaches for compositional safety and equivalence checking (e.g., [15,24,25]) which are not applicable for fault-resistance verification, our approach leverages the structural feature of round-based cryptographic circuits and decomposes the circuit into a set of single-round sub-circuits extended with, importantly, primary inputs/outputs, registers and their connections to guarantee soundness. We then verify those sub-circuits by leveraging SAT/SMT- and BDD-based approaches [31,37]. Our decomposition approach guarantees that the composition of fault-resistant single-round sub-circuits is always fault-resistant. Furthermore, we investigate various acceleration techniques that can significantly enhance verification efficiency.

We implement our approach as an open-source tool CLEAVE (**C**ompositional fau**L**t inj**E**ction **A**ttacks **VE**rifier), based on Verilog gate-level netlist. We thoroughly evaluate CLEAVE on 9 real-world cryptographic circuits (i.e., AES and LED64) equipped by both detection- and correction-based countermeasures, where the number of gates ranges from 1,020 to 34,351. The experimental results show that our approach is effective and efficient. For instance, the SAT-based compositional approach can verify most of the benchmarks (17/18) within 200 s and the remaining one can be done in 53 min; in contrast, the monolithic counterpart can only deal with 12 benchmarks within 6 h and requires significantly more verification time. The same improvements can be observed for SMT- and BDD-based compositional approaches.

To summarize, we make the following contributions.

- We propose a novel compositional fault-resistance verification framework for cryptographic circuits and various techniques to enhance efficiency;
- We implement an open-source tool CLEAVE for Verilog gate-level netlists;
- We extensively evaluate our tool on realistic cryptographic circuits, demonstrating its effectiveness and efficiency.

Outline. Section 2 introduces preliminaries. Section 3 defines the fault-resistance verification problem. Section 4 presents our compositional verification approach. Section 5 reports experimental results; We discuss related work in Sect. 6 and conclude the work in Sect. 7. Benchmarks, the source code of CLEAVE, more experiential results and missing proofs are provided [38].

2 Preliminaries

Let $\mathbb{B} := \{\mathtt{0}, \mathtt{1}\}$ and $[n] := \{1, \cdots, n\}$ for a natural number $n \geq 1$. We consider two types of logic gates: one-input gate $g : \mathbb{B} \to \mathbb{B}$ (e.g., `not`) and two-input gate $g : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (e.g., `and`, `or`, `xor`). To model faulty gates, we define three faulty counterparts ($\overline{g}$, g_s, g_r) of each gate g with $\overline{g} = \neg g$, $g_s = \mathtt{1}$ and $g_r = \mathtt{0}$.

Definition 1. *A combinational circuit C is a tuple (V, I, O, E, g), where*

- *V is a finite set of vertices in the circuit such that each vertex $v \in V \setminus (I \cup O)$ is associated with a logic gate $\mathsf{g}(v)$ whose fan-in is the in-degree of v;*
- *$I \subseteq V$ and $O \subseteq V$ are the primary inputs and outputs, respectively;*
- *$E \subseteq (V \setminus O) \times (V \setminus I)$ is a set of edges, each of which $(v_1, v_2) \in E$ transmits the signal over $\mathbb{B}$ from v_1 to v_2, namely, one of the inputs of the logic gate $\mathsf{g}(v_2)$ is driven by the output of the logic gate $\mathsf{g}(v_1)$;*
- *and (V, E) forms a Directed Acyclic Graph (DAG).*

A combinational circuit C represents a Boolean function $[\![C]\!] : \mathbb{B}^{|I|} \to \mathbb{B}^{|O|}$ such that for any input signals $\boldsymbol{x} \in \mathbb{B}^{|I|}$, $[\![C]\!](\boldsymbol{x})$ is the output of the circuit C when fed with $\boldsymbol{x}$.

A (synchronous) sequential circuit is a combinational circuit with feedback via registers and synchronized by a global clock. It is memoryful as the registers store the internal state. In this paper, we focus on *round-based* circuit implementations of cryptographic algorithms. Conceptually, the circuit consists of several rounds, and physically each round may comprise some clock cycles. For our purpose, the sequential circuit is defined as follows.

Definition 2. *A k-clock cycle sequential circuit $\mathcal{S}[k]$ (we may simply write $\mathcal{S}$ to simplify the notation) is a tuple $(\mathcal{I}, \mathcal{O}, \mathcal{C}, \mathcal{R}, \boldsymbol{s}_0)$, where*

- *$\mathcal{I}$ and $\mathcal{O}$ comprise the primary inputs and primary outputs, respectively.*
- *$\mathcal{R} = \mathcal{R}_{in} \cup \mathcal{R}_s$ is a finite set of registers (aka memory gates), with initial signals $\boldsymbol{s}_0 \in \mathbb{B}^{|\mathcal{R}_s|}$ for state registers in $\mathcal{R}_s$. Intuitively, registers in $\mathcal{R}_{in}$ (resp. $\mathcal{R}_s$) store primary input signals (resp. results) of combinational circuits.*
- *$\mathcal{C} = \{C_1, \cdots, C_k\}$, where for each $i \in [k]$, $C_i = (V_i, I_i, O_i, E_i, \mathsf{g}_i)$ is a combinational circuit for the i-th clock cycle. Moreover, it is required that all the primary inputs $\mathcal{I}$ are connected to registers in $\mathcal{R}_{in}$ which in turn are connected to the inputs I_i to avoid glitches, and the outputs O_i are connected to the primary outputs $\mathcal{O}$ and registers in $\mathcal{R}_s$. We also extend function g_i such that $\mathsf{g}_i(r)$ is an identity function for every register $r \in \mathcal{R}$*

A *state* $\boldsymbol{s} : \mathcal{R}_s \to \mathbb{B}$ of $\mathcal{S}[k]$ is a valuation of the registers $\mathcal{R}_s$. In each clock cycle $i \in [k-1]$, given a state $\boldsymbol{s}_{i-1}$ and primary input signals $\boldsymbol{x}_i$, the next state $\boldsymbol{s}_i$ is $[\![C_i]\!](\boldsymbol{s}_{i-1}, \boldsymbol{x}_i)$ projected onto $\mathcal{R}_s$, while $[\![C_i]\!](\boldsymbol{s}_{i-1}, \boldsymbol{x}_i)$ projected onto $\mathcal{O}$ gives the primary output signals $\boldsymbol{y}_i$, written as $\boldsymbol{s}_{i-1} \xrightarrow{\boldsymbol{x}_i|\boldsymbol{y}_i} \boldsymbol{s}_i$.

Given a sequence of primary input signals $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$, a *run* ρ of the circuit $\mathcal{S}[k]$ is a sequence

$$\boldsymbol{s}_0 \xrightarrow{\boldsymbol{x}_1|\boldsymbol{y}_1} \boldsymbol{s}_1 \xrightarrow{\boldsymbol{x}_2|\boldsymbol{y}_2} \boldsymbol{s}_2 \xrightarrow{\boldsymbol{x}_3|\boldsymbol{y}_3} \boldsymbol{s}_3 \longrightarrow \cdots \longrightarrow \boldsymbol{s}_{k-1} \xrightarrow{\boldsymbol{x}_k|\boldsymbol{y}_k} \boldsymbol{s}_k,$$

where $(\boldsymbol{y}_1, \cdots, \boldsymbol{y}_k)$ is the sequence of primary output signals. The circuit $\mathcal{S}[k]$ can also be seen as a Boolean function $[\![\mathcal{S}[k]]\!] : (\mathbb{B}^{|\mathcal{I}|})^k \to (\mathbb{B}^{|\mathcal{O}|})^k$ such that $[\![\mathcal{S}[k]]\!](\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ is the sequence of primary output signals for a sequence of primary input signals $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$.

We remark that our definition of sequential circuits is slightly different from the one given in [37], in which primary inputs can be connected to logic gates. We only allow primary inputs to connect to registers to avoid glitches which often introduce faults as well. Hence, our definition is sufficient for cryptographic circuits according to our experience while it facilitates the decomposition.

3 The Fault-Resistance Verification Problem

A fault injection attack actively injects faults into the execution of a cryptographic circuit and then infers sensitive data (such as the cryptographic key) via statistical analysis [3,8,9]. A general introduction refers to [21]. In particular, both non-invasive fault injections (i.e., clock glitches, underpowering and voltage glitches) and semi-invasive fault injections (i.e., electromagnetic pulses and laser beams) have been widely studied to compromise the security of cryptographic circuits, varying with attack cost and attack effectiveness [30]. There are detection- and correction-based countermeasures to mitigate fault injection attacks [1,35]: the former aims to detect fault injection attacks and raise an error flag once the attack is detected, so sensitive data can be destroyed in time; the latter aims to correct faults induced by attacks and produce the desired outputs.

3.1 Security Notions

We consider the following three fault types that suffice to capture both non-invasive fault injections and semi-invasive fault injections (cf. [30,37]):

– bit-set fault τ_s: when injected on a gate g, its output becomes $\mathtt{1}$, namely, the gate g becomes the faulty gate g_s, denoted by $\tau_s(g)$;
– bit-reset fault τ_r: when injected on a gate, its output becomes $\mathtt{0}$, namely, the gate g becomes the faulty gate g_r, denoted by $\tau_r(g)$;
– bit-flip fault τ_{bf}: when injected on a gate, its output is flipped, namely, the gate g becomes the faulty gate $\overline{g}$, denoted by $\tau_{bf}(g)$;

Fix a circuit $\mathcal{S}[k] = (\mathcal{I}, \mathcal{O}, \mathcal{R}, \boldsymbol{s}_0, \mathcal{C})$ protected using either a detection-based or correction-based countermeasure, where $\mathcal{C} = \{C_1, \cdots, C_k\}$ and for each $i \in [k]$, $C_i = (V_i, I_i, O_i, E_i, \mathsf{g}_i)$. We assume $o_{\mathtt{flag}} \in \mathcal{O}$, where $o_{\mathtt{flag}}$ is an error flag indicating whether a fault was detected when $\mathcal{S}$ adopts a detection-based countermeasure. If $\mathcal{S}$ adopts a correction-based countermeasure (i.e., no error flag is involved), we simply assume that $o_{\mathtt{flag}}$ is always $\mathtt{0}$. We denote by $\mathbf{B}$ the blacklist of invulnerable gates that are protected against fault injection attacks. $\mathbf{B}$ usually contains the gates used in implementing a countermeasure.

Definition 3. *A* fault vector *on the circuit $\mathcal{S}$ with the blacklist* $\mathbf{B}$ *and a set of fault types T, denoted by $\mathsf{V}(\mathcal{S}, \mathbf{B}, T)$, is a set of fault events*

$$\mathsf{V}(\mathcal{S}, \mathbf{B}, T) := \big\{\mathsf{e}(\alpha_1, \beta_1, \tau_1), \cdots, \mathsf{e}(\alpha_m, \beta_m, \tau_m) \mid i \neq j \implies (\sigma_i \neq \sigma_j \vee \beta_i \neq \beta_j)\big\},$$

where each fault event $\mathsf{e}(\sigma, \beta, \tau)$ consists of

- $\sigma \in [k]$ *specifying the clock cycle of the fault injection, namely, the fault injection occurs at the σ-th clock cycle;*
- $\beta \in \mathcal{R} \cup V_\sigma \setminus (I_\sigma \cup O_\sigma)$ *specifying the vulnerable gate on which the fault is injected (note that $\beta \notin \mathbf{B}$);*
- $\tau \in T$ *specifying the fault type.*

A fault vector $\mathsf{V}(\mathcal{S}, \mathbf{B}, T)$ yields a faulty circuit $\mathcal{F}(\mathcal{S}, \mathbf{B}, T) := (\mathcal{I}, \mathcal{O}, \mathcal{R}, \boldsymbol{s}_0, \mathcal{C}')$, where $\mathcal{C}' = \{C_1', \cdots, C_k'\}$, for each $i \in [k]$: $C_i' := (V_i, I_i, O_i, E_i, \mathsf{g}_i')$ and $\mathsf{g}_i'(\beta) := \tau(\mathsf{g}_i(\beta))$ if $\mathsf{e}(i, \beta, \tau) \in \mathsf{V}(\mathcal{S}, \mathbf{B}, T)$, otherwise $C_i' := C_i$ and $\mathsf{g}_i'(\beta) := \mathsf{g}_i(\beta)$.

Intuitively, the faulty circuit $\mathcal{F}(\mathcal{S}, \mathbf{B}, T)$ is the same as the circuit $\mathcal{S}$ except that for each fault event $\mathsf{e}(i, \beta, \tau) \in \mathsf{V}(\mathcal{S}, \mathbf{B}, T)$, the gate $\mathsf{g}_i(\beta)$ is transiently replaced by its faulty counterpart $\tau(\mathsf{g}_i(\beta))$ in the i-th clock cycle, whereas all the other gates remain the same.

Definition 4. *A fault vector $\mathsf{V}(\mathcal{S}, \mathbf{B}, T)$ is* effective *if there exists a sequence of primary input signals $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ such that two sequences of primary output signals*

$$[\![\mathcal{S}]\!](\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k) \text{ and } [\![\mathcal{F}(\mathcal{S}, \mathbf{B}, T)]\!](\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$$

differ at some clock cycle before the error flag $o_{\mathtt{flag}}$ is set.

Otherwise, the fault vector $\mathsf{V}(\mathcal{S}, \mathbf{B}, T)$ is ineffective *and the circuit $\mathcal{S}$ is* resistant *against the fault vector $\mathsf{V}(\mathcal{S}, \mathbf{B}, T)$.*

An effective fault vector results in faulty primary output signals where the fault is *not* successfully detected (i.e., the error flag $o_{\mathtt{flag}}$ is not set in time). Note that there are two possible cases for an ineffective fault vector: either $[\![\mathcal{S}]\!](\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ and $[\![\mathcal{F}(\mathcal{S}, \mathbf{B}, T)]\!](\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ are the same or the fault is successfully detected.

Inspired by the consolidated fault model [30], we define the security model for fault-resistance verification which characterizes the capabilities of the adversary.

Definition 5. *A* fault-resistance model *for the circuit $\mathcal{S}$ with the blacklist* $\mathbf{B}$ *is given by $\mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$, where*

- $\mathtt{n}_e$ *is the maximum number of fault events per clock cycle;*
- $\mathtt{n}_c$ *is the maximum number of clock cycles in which fault events can occur;*
- $T \subseteq \{\tau_s, \tau_r, \tau_{bf}\}$ *specifies the set of allowed fault types; and*
- $\ell \in \{\mathtt{c}, \mathtt{r}, \mathtt{cr}\}$ *defines vulnerable gates:* $\mathtt{c}$ *for logic gates in combinational circuits,* $\mathtt{r}$ *for registers and* $\mathtt{cr}$ *for both logic gates and registers.*

For example, $\mathfrak{m}(\mathtt{n}_e, k, \{\tau_s, \tau_r, \tau_{bf}\}, \mathtt{cr})$ models the strongest adversary, who can inject faults to all the gates simultaneously at any clock cycle (except for those protected in the blacklist $\mathbf{B}$) while $\mathfrak{m}(1, 1, \{\tau_s\}, \mathtt{c})$ only allows the adversary to choose one logic gate to inject a set fault in one chosen clock cycle.

Formally, the fault-resistance model $\mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$ defines the following set $[\![\mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)]\!]$ of possible fault vectors that can be applied by the adversary:

$$[\![\mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)]\!] := \left\{ \mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T) \;\middle|\; \begin{array}{c} \sharp\mathtt{MaxE}(\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)) \leq \mathtt{n}_e \\ \text{and} \\ \sharp\mathtt{Clk}(\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)) \leq \mathtt{n}_c \end{array} \right\}$$

where

- $\mathbf{B}_\ell := \begin{cases} \mathbf{B}, & \text{if } \ell = \mathtt{cr}; \\ \mathbf{B} \cup \mathcal{R}, & \text{if } \ell = \mathtt{c}; \\ \mathbf{B} \cup \bigcup_{i \in [k]} V_i \setminus (I_i \cup O_i), & \text{if } \ell = \mathtt{r}; \end{cases}$
- $\sharp\mathtt{MaxE}(\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)) := \max_{\alpha \in [k]} |\{\mathsf{e}(\alpha, \beta, \tau) \in \mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)\}|$, i.e., the maximum number of fault events per clock cycle in the fault vector $\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)$;
- $\sharp\mathtt{Clk}(\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)) := |\{\alpha \mid \mathsf{e}(\alpha, \beta, \tau) \in \mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)\}|$, i.e., the number of clock cycles when fault events can occur.

Definition 6. *The circuit $\mathcal{S}$ is* fault-resistant *against $\mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$, denoted by $\langle \mathcal{S}, \mathbf{B} \rangle \models \mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$, if all the fault vectors $\mathsf{V}(\mathcal{S}, \mathbf{B}, T) \in [\![\mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)]\!]$ are ineffective.*

The fault-resistance verification problem *is to determine whether or not $\langle \mathcal{S}, \mathbf{B} \rangle \models \mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$.*

By Definition 6, it is straightforward to show that:

Proposition 1. *If $\langle \mathcal{S}, \mathbf{B} \rangle \models \mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T_1, \mathtt{cr})$, then $\langle \mathcal{S}, \mathbf{B}' \rangle \models \mathfrak{m}(\mathtt{n}'_e, \mathtt{n}'_c, T_2, \ell)$ for any $\mathbf{B} \subseteq \mathbf{B}'$, $\mathtt{n}'_e \leq \mathtt{n}_e$, $\mathtt{n}'_c \leq \mathtt{n}_c$, $T_2 \subseteq T_1$, $\ell \in \{\mathtt{c}, \mathtt{r}, \mathtt{cr}\}$.*

By adapting the proof of NP-completeness [37] which reduces from the SAT problem, we can show

Theorem 1. *The problem of determining whether a k-clock cycle circuit $\mathcal{S}[k]$ for any fixed $k \geq 3$ is not fault-resistant is NP-complete.*

3.2 Motivating Example

A motivating example is given in Fig. 1, which is a simplified implementation of AES with a detection-based countermeasure [1]. The circuit has three cryptographic blocks (`B1`, `B2`, `B3`), three redundancy blocks (`RB1`, `RB2`, `RB3`), two selective blocks (`MUX1`, `MUX2`) and a check block `CHECK`, where all the gates in the check block `CHECK` are added to the blacklist $\mathbf{B}$. The cryptographic blocks and the two selective blocks together implement the functionality of AES, while the others implement a detection-based countermeasure.

The *first round* starts with a reset signal `rst` (i.e., `rst` =`1`) after which the primary input signals `INPUT` are selected by `MUX1` and stored in the registers `REG`. Moreover, `rst` is set to `0`. Next, the values stored in the registers `REG` are processed by the cryptographic and redundancy blocks. The cryptographic block `B1` produces primary output signals of the current round; the results of the cryptographic block `B3` and redundancy block `RB3` are stored in the registers `REG` as inputs of the next round (called feedback). Furthermore, the values of registers and the results of all the cryptographic and redundancy blocks are fed to the check block `CHECK` which checks whether a fault injection attack occurs. The primary output `FLAG` is the error flag.

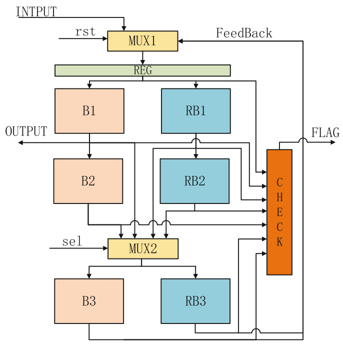


Fig. 1. The AES circuit.

The *internal rounds* are the same as the first round except that the feedback from the previous round is stored in the registers, instead of the primary input signals, because the reset signal `rst` has been set to `0` in the first round. The *last round* is the same as the internal rounds except that the results of the cryptographic block `B1` (resp. the redundancy block `RB1`) are fed to the cryptographic block `B3` (resp. the redundancy block `RB3`) by setting the input signal `sel`=`1` of the selective block `MUX2`, respectively.

To verify its fault-resistance, one can unroll it according to the clock cycle (cf. [38]), then enumerate and check the effectiveness of each possible fault vector by analyzing the unrolled and faulty counterparts via BDD [31] or SAT/SMT [37]. However, there are two shortcomings which hurdle their efficiency and scalability. (1) One shall verify the equivalence of the primary outputs of the circuit and its faulty counterpart, which must be done for each round (unless the error flag is set). Since the subsequent round depends upon preceding rounds, the size of the SAT/SMT formulas or BDDs usually increases dramatically, which incurs a blowup in rounds of circuits. (2) To achieve completeness (or at least a high coverage), a large number of possible fault vectors have to be checked, which incurs a blowup in the number of fault vectors. Our work proposes a novel compositional approach to combat these two types of blowups in fault-resistance verification by decomposing the verification of an entire circuit into the verification of (typically much smaller) single-round sub-circuits.

4 Compositional Verification

In this section, we first describe the overview of our approach and our decomposition, next briefly recap two symbolic approaches (SAT/SMT- and BDD-based) for verifying sub-circuits, and finally present three acceleration techniques to improve the verification efficiency.

4.1 Overview of the Approach

Our approach relies on the structural feature of (round-based) cryptographic primitives, e.g., block ciphers, for which countermeasures are developed round-by-round accordingly, aiming to isolate the effects of fault injection in each round. Furthermore, the rounds are often similar, many of which are even the same, For instance, the first $(k-1)$ rounds in Fig. 1 are the same except that the first round uses the primary input signals while the other (internal) rounds use the feedback from the previous round (i.e., the values stored in the registers).

Based on the above key observation, as shown in Fig. 2, given a circuit $\mathcal{S}$, a blacklist $\mathbf{B}$ of gates on which faults cannot be injected and a fault-resistance model $\mathfrak{m}(n_e, n_c, T, \ell)$, our approach first decomposes the circuit $\mathcal{S}$ into single-round sub-circuits $(S_1, \cdots, S_r)$ where each S_i for $i \in [r]$ implements one round. As many sub-circuits are indeed identical, we only need to verify a small number of single-round sub-circuits in isolation whereby the fault-resistance of the entire circuit $\mathcal{S}$ is guaranteed. For instance, in the motivating example, we only need to verify the first and the k-th (i.e., last) round, because the first $(k-1)$ rounds are virtually the same. It reduces the verification of a k-round circuit to the verification of two single-round sub-circuits.

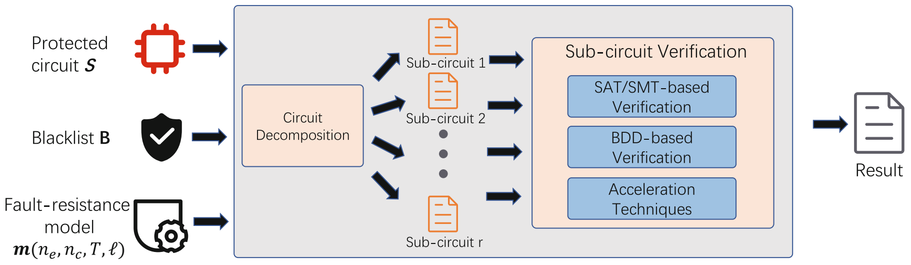


Fig. 2. Overview of our approach.

To verify each sub-circuit, we leverage two symbolic verification approaches, based on SAT/SMT and BDD. To further improve efficiency, we also study various acceleration techniques exploiting fault effects and propagation.

4.2 The Decomposition

For a k-clock cycle circuit $\mathcal{S}[k] = (\mathcal{I}, \mathcal{O}, \mathcal{R}, \mathbf{s}_0, \mathcal{C})$ where $\mathcal{R} = \mathcal{R}_{in} \cup \mathcal{R}_s$, $\mathcal{C} = \{C_1, \cdots, C_k\}$ and $C_i = (V_i, I_i, O_i, E_i, \mathsf{g}_i)$ for each $i \in [k]$, let r be the number of rounds of $\mathcal{S}[k]$. An *r-decomposition* of $\mathcal{S}[k]$ is $(S_1[k_1], \cdots, S_r[k_r])$, where for every $i \in [r]$, $S_i[k_i]$ is a *single-round, k_i-clock cycle sub-circuit* $(\mathcal{I}^{(i)}, \mathcal{O}^{(i)}, \mathcal{R}^{(i)}, \mathbf{s}^{(i)}, \mathcal{C}^{(i)})$ defined as (note that $\sum_{i\in[r]} k_i = k$)

- $\mathcal{I}^{(i)} = \mathcal{I} \cup \mathcal{I}_{fb}$, where $\mathcal{I}_{fb}$ comprises additional primary inputs used for representing the signals passed from the previous round, i.e., the values stored in the state registers $\mathcal{R}_s$ at the end of the $(i-1)$-th round;
- $\mathcal{O}^{(i)} = \mathcal{O} \cup \mathcal{O}_{fb}$, where $\mathcal{O}_{fb}$ comprises additional primary outputs used for representing the signals passed to the next round, i.e., the values stored to the state registers $\mathcal{R}_s$ at the end of the $(i-1)$-th round;
- $\mathcal{R}^{(i)} = \mathcal{R}'_{in} \cup \mathcal{R}'_s$ where $\mathcal{R}'_{in} = \mathcal{R}_{in} \cup \mathcal{R}_s^{in}$, $\mathcal{R}_s^{in} \subseteq \mathcal{R}_s$ comprises registers used for storing signals passed from one round to the next round, and $\mathcal{R}'_s \subseteq \mathcal{R}_s$ comprises the registers used for connecting combinational circuits of $\mathcal{C}^{(i)}$ (note that $\mathcal{R}'_s$ can be $\emptyset$ if $k_i = 1$, i.e., the round has one clock cycle);
- $\boldsymbol{s}^{(1)} = \boldsymbol{s}_0$ and $\boldsymbol{s}^{(i)}$ for $i \geq 2$ is not defined;
- $\mathcal{C}^{(i)} = \{C_{i,1}, \cdots, C_{i,k_i}\}$ with $C_{1,1}, \cdots, C_{1,k_1}, \cdots, C_{r,1}, \cdots, C_{r,k_r} = C_1 \cdots C_k$, and the connection between any two adjacent single-rounds sub-circuits via the registers $\mathcal{R}'_s$ is the same as that in $\mathcal{S}$;
- the registers in $\mathcal{R}_s^{in}$ that were connected by the outputs $O_{i-1,k_{i-1}}$ of $C_{i-1,k_{i-1}}$ are now connected by the additional primary inputs $\mathcal{I}_{fb}$ if $i \geq 2$;
- the outputs $O_{i-1,k_{i-1}}$ of C_{i,k_i} that were connected to the registers in $\mathcal{R}_s$ are now connected to the additional primary outputs $\mathcal{O}_{fb}$.

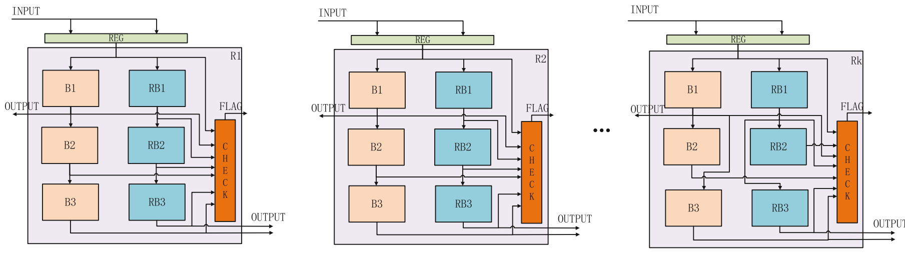


Fig. 3. Single-round sub-circuits of the motivating example.

Two single-round sub-circuits $S_i[k_i]$ and $S_j[k_j]$ are *isomorphic* w.r.t. the blacklist $\mathbf{B}$ if they are identical up to the renaming of the primary inputs/outputs, registers and vertices in the combinational circuits, and the matched gate pairs are either both protected or not protected in $\mathbf{B}$. Note that this condition is much stricter than the semantic equivalence of two circuits, namely, the same input-output relation, which is insufficient for our decomposition theorem. For instance, consider one single-round sub-circuit correctly implements a correction-based countermeasure but the other one does not implement any countermeasure. They are semantically equivalent, but both have to be verified.

Proposition 2. *For any pair of isomorphic circuits* (S_i, S_j) *and fault-resistance model* $\mathtt{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$, $\langle S_i, \mathbf{B}\rangle \models \mathtt{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$ *iff* $\langle S_j, \mathbf{B}\rangle \models \mathtt{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$. □

Consider the example in Fig. 1. In this case, $r = k$ ($k_i = 1$ for each $i \in [k]$). As illustrated in Fig. 3, our r-decomposition removes all the connections labeled with

FeedBack, re-connects the outputs of the blocks B3 and RB3 to the additional primary outputs that were connected to the registers REG, and connects the additional primary inputs to the registers REG that were connected by the outputs from the previous round. Then, all the single-sound sub-circuits except for the last one are isomorphic.

Theorem 2. *Given a k-clock cycle circuit $\mathcal{S}[k] = (\mathcal{I}, \mathcal{O}, \mathcal{R}, \boldsymbol{s}_0, \mathcal{C})$ and a blacklist $\mathbf{B}$, let $(S_1[k_1], S_2[k_2], \cdots, S_r[k_r])$ be the r-decomposition of $\mathcal{S}[k]$. For any fault-resistance model $\mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$, if $\langle S_i, \mathbf{B}\rangle \models \mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$ for all single-round sub-circuits $S_i \in \{S_1, S_2 \cdots, S_r\}$, then $\langle \mathcal{S}, \mathbf{B}\rangle \models \mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$.*

Furthermore, if $\mathtt{n}_c \geq k_i$ for all $i \in [r]$,then $\langle \mathcal{S}, \mathbf{B}\rangle \models \mathfrak{m}(\mathtt{n}_e, k, T, \ell)$.

We should emphasize that the additional primary inputs $\mathcal{I}_{fb}$, primary outputs $\mathcal{O}_{fb}$, registers $\mathcal{R}_s^{in}$ and their connections are crucial to guarantee that the composition $\mathcal{S}[k]$ of the fault-resistant sing-round sub-circuits $(S_1[k_1], \cdots, S_r[k_r])$ is also fault-resistant. The fault-resistance of all the single-round sub-circuits, i.e., $\langle S_i, \mathbf{B}\rangle \models \mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$ for $i \in [r]$, ensures that the primary outputs $\mathcal{O}' = \mathcal{O} \cup \mathcal{O}_{bf}$ remain the same (unless the error flag is set) for any fault vector $\mathsf{V}(\mathcal{S}, \mathbf{B}, T) \in \mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$. It guarantees that not only the primary outputs $\mathcal{O}$ but also the values stored to the registers $\mathcal{R}_s^{in}$ at the end of each round remain the same (unless the error flag is set) for any fault vector $\mathsf{V}(\mathcal{S}, \mathbf{B}, T) \in \mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$. In other words, the single-round sub-circuits are able to detect any fault injections which change the primary outputs $\mathcal{O}$ or the values used by the next round (i.e., isolating fault effects in each round). Thus, our decomposition approach for compositional fault-resistance verification is different from previous ones used for compositional safety and equivalence checking (e.g., [15,24,25]).

4.3 SAT/SMT-Based Verification

We adopt the SAT/SMT-based approach used in FIRMER [37] which reduces the problem to SAT/SMT solving. Given a fault-resistance model $\mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$ and a (single-round) k-clock cycle circuit $\mathcal{S}[k] = (\mathcal{I}, \mathcal{O}, \mathcal{R}, \boldsymbol{s}_0, \mathcal{C})$, FIRMER first encodes all the possible fault vectors into $\mathcal{S}[k]$ by introducing additional inputs to control if a fault is injected on a gate and which fault type is injected. This will result in a controllable faulty circuit, denoted by $\mathcal{S}_{\mathfrak{m}}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$. The fault-resistance verification of $\mathcal{S}[k]$ is reduced to equivalence checking of $\mathcal{S}$ and $\mathcal{S}_{\mathfrak{m}}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$ with constraints on the additional inputs and error flag, which in turn is reduced to the SAT/SMT solving. (Cf. [37] for details.)

4.4 BDD-Based Verification

We adopt the BDD-based approach used in FIVER [31]. To avoid re-construction of the BDD from scratch for each fault vector, FIVER first attaches each gate g in the circuit $\mathcal{S}$ with a BDD D_g representing the output of the gate in $\mathcal{S}$. Then, for each fault vector $\mathsf{V}(\mathcal{S}, \mathbf{B}, T) \in [\![\mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)]\!]$, on a copy $\mathcal{S}'$ of the

BDD-attached circuit $\mathcal{S}$, the BDD D_g of the gate g is revised according to each fault event $\mathsf{e}(i, g, \tau) \in \mathsf{V}(\mathcal{S}, \mathbf{B}, T)$, where the BDDs of the gates depending upon g are also revised accordingly. Finally, for each clock cycle, FIVER checks each primary output o by comparing the attached BDDs of the primary output o in the circuit $\mathcal{S}$ and its faulty counterpart $\mathcal{S}'$. Furthermore, some optimizations to reduce the number of considered fault vectors and improve the construction of the desired $\mathcal{S}'$ are implemented. (Cf. [31] for details.)

4.5 Acceleration Techniques

For both SAT/SMT-based and BDD-based verification, we apply the following acceleration techniques.

Fixed Number of Fault Events. Recall that to prove fault-resistance, we considered all possible fault vectors $\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)$ such that $\sharp\texttt{MaxE}(\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)) \leq \texttt{n}_e$ and $\sharp\texttt{Clk}(\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)) \leq \texttt{n}_c$. It turns out that these two conditions can be safely improved to "$\texttt{n}_e$ fault events for each clock cycle if some fault events occur in this clock cycle" when $\tau_s, \tau_r \in T$ and the number of vulnerable gates is more than $\texttt{n}_e$ in each clock cycle, reducing the number of fault vectors to be checked. Indeed, if there is an effective fault vector $\mathsf{V}(\mathcal{S}, \mathbf{B}, T) \in [\![\mathfrak{m}(\texttt{n}_e, \texttt{n}_c, T, \ell)]\!]$ such that the number of fault events is n in some clock cycle with $1 \leq n < \texttt{n}_e$, there exists a sequence of primary input signals $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ such that $[\![\mathcal{S}]\!](\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ and $[\![\mathcal{F}(\mathcal{S}, \mathbf{B}, T)]\!](\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ differ at some clock cycle before the error flag is set. We can add $(\texttt{n}_e - n)$ fault events $\mathsf{e}(i, g, \tau)$ to $\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)$, where the output of the gate g under the primary input signals $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ remains the same by choosing $\tau \in \{\tau_s, \tau_r\}$. The resulting fault vector is still effective.

Fault Type Reduction. Let $\mathcal{T} = \{\tau_s, \tau_r, \tau_{bf}\}$. We find that $\langle \mathcal{S}, \mathbf{B} \rangle \models \mathfrak{m}(\texttt{n}_e, \texttt{n}_c, \mathcal{T}, \ell)$ iff $\langle \mathcal{S}, \mathbf{B} \rangle \models \mathfrak{m}(\texttt{n}_e, \texttt{n}_c, \tau_{bf}, \ell)$ iff $\langle \mathcal{S}, \mathbf{B} \rangle \models \mathfrak{m}(\texttt{n}_e, \texttt{n}_c, \{\tau_s, \tau_r\}, \ell)$, allowing us to consider only $\{\tau_s, \tau_r\}$ if $\{\tau_s, \tau_r\} \subseteq T$ and only τ_{bf} if $\tau_{bf} \in T$ for any set T of fault types. Consider an effective fault vector $\mathsf{V}(\mathcal{S}, \mathbf{B}, \mathcal{T}) \in [\![\mathfrak{m}(\texttt{n}_e, \texttt{n}_c, \mathcal{T}, \ell)]\!]$ and a sequence of primary input signals $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ such that $[\![\mathcal{S}]\!](\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ and $[\![\mathcal{F}(\mathcal{S}, \mathbf{B}, \mathcal{T})]\!](\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ differ at some clock cycle before the error flag is set.

- For every fault event $\mathsf{e}(i, g, \tau_{bf}) \in \mathsf{V}(\mathcal{S}, \mathbf{B}, \mathcal{T})$, if the output of the gate g at the i-th clock cycle in $[\![\mathcal{F}(\mathcal{S}, \mathbf{B}, \mathcal{T})]\!](\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ is flipped from $\texttt{1}$ to $\texttt{0}$ (resp. from $\texttt{0}$ to $\texttt{1}$), $\mathsf{e}(i, g, \tau_{bf})$ can be safely replaced by $\mathsf{e}(i, g, \tau_r)$ (resp. $\mathsf{e}(i, g, \tau_s)$). Thus, $\langle \mathcal{S}, \mathbf{B} \rangle \models \mathfrak{m}(\texttt{n}_e, \texttt{n}_c, \{\tau_s, \tau_r\}, \ell)$ entails $\langle \mathcal{S}, \mathbf{B} \rangle \models \mathfrak{m}(\texttt{n}_e, \texttt{n}_c, \mathcal{T}, \ell)$.
- For every fault event $\mathsf{e}(i, g, \tau) \in \mathsf{V}(\mathcal{S}, \mathbf{B}, \mathcal{T})$ such that $\tau \in \{\tau_s, \tau_r\}$, if the output of the gate g at the i-th clock cycle in $[\![\mathcal{F}(\mathcal{S}, \mathbf{B}, \mathcal{T})]\!](\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ is flipped by applying $\mathsf{e}(i, g, \tau)$, $\mathsf{e}(i, g, \tau)$ can be safely replaced by $\mathsf{e}(i, g, \tau_{bf})$; otherwise the output of the gate g at the i-th clock cycle in $[\![\mathcal{F}(\mathcal{S}, \mathbf{B}, \mathcal{T})]\!](\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$ remains the same by applying $\mathsf{e}(i, g, \tau)$, $\mathsf{e}(i, g, \tau)$ can be safely removed from $\mathsf{V}(\mathcal{S}, \mathbf{B}, \mathcal{T})$. Thus, $\langle \mathcal{S}, \mathbf{B} \rangle \models \mathfrak{m}(\texttt{n}_e, \texttt{n}_c, \tau_{bf}, \ell)$ entails $\langle \mathcal{S}, \mathbf{B} \rangle \models \mathfrak{m}(\texttt{n}_e, \texttt{n}_c, \mathcal{T}, \ell)$.

Vulnerable Gate Reduction. If the output of a gate g is *only* connected to one vulnerable logic gate $g' \notin \mathbf{B}_\ell$, then the gate g can be safely added into the blacklist $\mathbf{B}$ while no protection is required for the gate g. It is because:

– if the output of the gate g does not change at the i-th clock cycle after applying the fault event $\mathtt{e}(i, g, \tau)$, then the effect of the fault event $\mathtt{e}(i, g, \tau)$ terminates at the gate g', thus $\mathtt{e}(i, g, \tau)$ can be removed from any fault vector;
– if the output of the gate g does change at the i-th clock cycle after applying the fault event $\mathtt{e}(i, g, \tau)$, it is flipped either from `1` to `0` or from `0` to `1`, the same effect can be achieved by applying the fault event $\mathtt{e}(i, g', \tau_{bf})$, or the fault event $\mathtt{e}(i, g', \tau_s)$ if it is flipped from `0` to `1` or the fault event $\mathtt{e}(i, g', \tau_r)$ if it is flipped from `1` to `0`.

As a result, it suffices to consider fault injections on the gate g' instead of both g and g' when $\tau_{bf} \in T$ or $\{\tau_s, \tau_r\} \subseteq T$, which reduces the number of vulnerable gates [37]. By a graph traversal of the circuit $\mathcal{S}$, all the gates g whose output is *only* connected to one vulnerable logic gate $g' \notin \mathbf{B}_\ell$ can be identified and then added into the blacklist $\mathbf{B}$.

We finally remark that the above three acceleration techniques can be applied simultaneously except that we cannot fix the number of fault events if the set and reset fault types (i.e., τ_s and τ_r) are unavailable.

5 Implementation and Evaluation

We have implemented our approach as an open-source tool CLEAVE based on the parallel SAT solver Glucose 4.2.1 [6] and SMT solver bitwuzla 1.0-prerelease [28], where the BDD-based compositional verification is implemented based on FIVER which uses the CUDD package. Given a circuit $\mathcal{S}$ in Verilog gate-level netlist, a blacklist $\mathbf{B}$ and a fault-resistance model $\mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$, CLEAVE determines whether $(\mathcal{S}, \mathbf{B}) \models \mathfrak{m}(\mathtt{n}_e, \mathtt{n}_c, T, \ell)$. Currently, CLEAVE directly extracts single-round sub-circuits from $\mathcal{S}$ by enumerating all the feasible combinations of input signals of selective blocks. One feasible combination gives one single-round sub-circuit on which fault resistance is verified. Though more than one isomorphic single-round sub-circuits may be verified, the computational-expensive (GI-complete) isomorphism checking of pairs of single-round sub-circuits is avoided. For instance, the two distinct single-round sub-circuits of the circuit in Fig. 1) are extracted by fixing the signals of `rst` and `sel` to $(\mathtt{1}, \mathtt{0})$, $(\mathtt{0}, \mathtt{0})$ and $(\mathtt{0}, \mathtt{1})$, respectively, where the first two pairs of signals give the same single-round sub-circuits after adding/re-connecting primary inputs/outputs and registers according to our decomposition.

Benchmarks. We use 9 VHDL implementations [1,35] of 3 cryptographic algorithms (i.e., CRAFT, LED and AES [31]). The VHDL implementations are transformed into Verilog gate-level netlists using the Synopsys design compiler (version O-2018.06-SP2). The blacklists are generated according to [1,35]. The statistics of the benchmarks are given in Table 1. The first column shows the name of the cryptographic algorithm, the maximal number of protected faulty bits per clock cycle (bi), the type of the adopted countermeasure (D for detection-based and C for correction-based). The second column shows the single-round sub-circuit and its number of times used in the implementation, e.g., the 10-round

AES-b1-D has two single-round sub-circuits (S1, S2) and S1 is used in 9 rounds. The other columns respectively give the size of the blacklist **B**, the numbers of primary inputs, primary outputs, gates and each specific gate.

We can observe that CRAFT benchmarks use both detection-based (D) and correction-based (C) countermeasures, many single-round sub-circuits are isomorphic in each implementation, the number of distinct single-round sub-circuits ranges from 1 to 3, and the number of gates in one single-round sub-circuit ranges from 1,020 to 34,351 so that the scalability of CLEAVE can be evaluated.

Setup. The experiments were conducted on a machine with Intel Xeon Gold 6342 2.80 GHz CPU, 1T RAM, and Ubuntu 20.04.1. Each verification task is run with 6-hour timeout. All the SAT-based and BDD-based (compositional) verification approaches are run with eight threads while the SMT-based (compositional) verification approaches are run with a single thread, with their default parameters (There are no promising parallel SMT solvers for QF_BV). The verification time is given in seconds with the best one highlighted in **boldface**, column R reports the verification result, and column DR shows the desired verification result. Mark ✓ (resp. ✗) indicates that the circuit is fault-resistant (resp. not fault-resistant) w.r.t. the fault-resistance model.

5.1 Effectiveness of Acceleration Techniques

Recall that we present three acceleration techniques: fixed number of fault events (denoted by FE), fault type reduction (denoted by TR), and vulnerable gate reduction (denoted by GR). We denote by "no-opt" the verification without any of these acceleration techniques, by TR_{sr} and TR_{bf} the fault type reduction that reduces to the fault types (τ_s, τ_r) and the fault type τ_{bf}, respectively.

Table 1. Benchmark statistics.

Name	Rnd	#Clk	|**B**|	#in	#out	#gate	#and	#nand	#or	#nor	#xor	#xnor	#not	#reg
AES-b1-D	S1×9	1	432	256	129	25,008	576	9,446	560	9,705	828	852	2,897	144
	S2×1	1	432	256	129	24,192	544	9,018	624	9,381	816	992	2,673	144
AES-b2-D	S1×9	1	1,055	256	129	34,351	704	12,698	833	13,012	1,440	1,584	3,888	192
	S2×1	1	1,055	256	129	33,423	752	12426	849	12,308	1,392	1,808	3,696	192
CRAFT-b1-D	S1×32	2	240	128	65	1,020	48	202	48	149	212	232	49	80
CRAFT-b2-D	S1×32	2	575	128	65	1,715	65	255	48	271	188	680	96	112
CRAFT-b3-D	S1×32	2	767	128	65	2,111	64	346	65	292	224	896	96	128
CRAFT-b1-C	S1×32	2	2,304	128	64	3,172	0	864	48	656	428	760	304	112
CRAFT-b2-C	S1×32	2	19,568	128	64	20,884	320	7,904	352	6,592	1,484	2,056	2,000	176
LED64-b1-D	S1×1	1	239	128	65	1,632	16	346	32	53	416	608	81	80
	S2×8	1	240	128	65	1,636	16	346	32	53	420	604	85	80
	S3×23	1	240	128	65	1,480	16	346	32	53	352	544	57	80
LED64-b2-D	S1×1	1	575	128	65	2,575	17	479	64	111	512	1168	112	112
	S2×8	1	575	128	65	2,585	17	479	64	111	516	1164	122	112
	S3×23	1	575	128	65	2,333	17	479	64	111	448	1024	78	112

Table 2. SAT-based verification of single-round sub-circuits.

Name	Model	no-opt	GR	GR·FE	GR·TR$_{sr}$	GR·FE·TR$_{sr}$	GR·TR$_{bf}$	R	DR
AES-b1-D	$\mathtt{m}(1,1,\mathcal{T},\mathtt{cr})$	2,486.33	255.06	219.26	197.15	214.07	**178.58**	✓	✓
AES-b1-D	$\mathtt{m}(2,1,\mathcal{T},\mathtt{cr})$	2.62	0.81	0.72	**0.60**	0.63	0.65	✗	✗
AES-b2-D	$\mathtt{m}(2,1,\mathcal{T},\mathtt{cr})$	timeout	2,409.43	2,272.56	2,224.11	2,412.66	**1,595.51**	✓	✓
AES-b2-D	$\mathtt{m}(3,1,\mathcal{T},\mathtt{cr})$	4.68	1.34	**0.94**	0.99	1.07	1.43	✗	✗
CRAFT-b2-C	$\mathtt{m}(2,1,\mathcal{T},\mathtt{cr})$	31.80	10.08	10.78	10.95	11.09	**9.40**	✓	✓
CRAFT-b2-C	$\mathtt{m}(3,1,\mathcal{T},\mathtt{cr})$	0.32	**0.26**	0.35	0.33	0.32	0.30	✗	✗
CRAFT-b3-D	$\mathtt{m}(3,1,\mathcal{T},\mathtt{cr})$	7.56	0.33	0.32	0.32	**0.31**	0.42	✓	✓
CRAFT-b3-D	$\mathtt{m}(4,1,\mathcal{T},\mathtt{cr})$	0.08	**0.04**	**0.04**	**0.04**	0.05	0.05	✗	✗

The acceleration techniques can be combined, e.g., GR·FE applies "vulnerable gate reduction" with "fixed number of fault events". Note that TR$_{bf}$ cannot be combined with a fixed number of fault events (i.e., no FE·TR$_{bf}$ or GR·FE·TR$_{bf}$). We evaluate all the acceleration techniques and their feasible combinations on the first single-round sub-circuits of AES-b1-D, AES-b2-D, CRAFT-b2-C, and CRAFT-b3-D.

The results of SAT-based verification are reported in Table 2. Overall, all three acceleration techniques and their combinations can improve the SAT-based verification approach (no-opt) for almost all the verification tasks, solving one timeout case and significantly reducing the verification time for the other cases. The combination GR·TR$_{bf}$ outperforms the others because encod-

Table 3. Results of fault-resistance verification: compositional vs. monolithic.

Name	Model	Compositional			Monolithic			R	DR
		BDD	SAT	SMT	BDD	SAT	SMT		
AES-b1-D	$\mathtt{m}(1,1,\mathcal{T},\mathtt{cr})$	**173.06**	193.49	15,944.55	timeout	timeout	timeout	✓	✓
AES-b1-D	$\mathtt{m}(2,1,\mathcal{T},\mathtt{cr})$	409.31	**1.65**	5,735.58	timeout	timeout	timeout	✗	✗
AES-b2-D	$\mathtt{m}(2,1,\mathcal{T},\mathtt{cr})$	timeout	**3,175.90**	timeout	timeout	timeout	timeout	✓	✓
AES-b2-D	$\mathtt{m}(3,1,\mathcal{T},\mathtt{cr})$	timeout	**2.25**	timeout	timeout	timeout	timeout	✗	✗
CRAFT-b1-C	$\mathtt{m}(1,1,\mathcal{T},\mathtt{cr})$	**0.13**	0.31	2.07	timeout	10,587.20	timeout	✓	✓
CRAFT-b1-C	$\mathtt{m}(2,1,\mathcal{T},\mathtt{cr})$	0.24	0.05	**0.04**	timeout	510.55	timeout	✗	✗
CRAFT-b2-C	$\mathtt{m}(2,1,\mathcal{T},\mathtt{cr})$	**3.02**	10.04	99.47	timeout	timeout	timeout	✓	✓
CRAFT-b2-C	$\mathtt{m}(3,1,\mathcal{T},\mathtt{cr})$	4.26	**0.38**	1.73	timeout	timeout	timeout	✗	✗
CRAFT-b1-D	$\mathtt{m}(1,2,\mathcal{T},\mathtt{cr})$	0.86	**0.13**	0.56	timeout	144.46	1,000.45	✓	✓
CRAFT-b1-D	$\mathtt{m}(2,2,\mathcal{T},\mathtt{cr})$	37.32	0.03	**0.02**	timeout	12.69	1.26	✗	✗
CRAFT-b2-D	$\mathtt{m}(2,2,\mathcal{T},\mathtt{cr})$	3,188.87	**0.30**	1.91	timeout	137.70	9,943.49	✓	✓
CRAFT-b2-D	$\mathtt{m}(3,2,\mathcal{T},\mathtt{cr})$	3,295.12	**0.04**	**0.04**	timeout	40.53	1.87	✗	✗
CRAFT-b3-D	$\mathtt{m}(3,2,\mathcal{T},\mathtt{cr})$	timeout	**0.44**	11.33	timeout	203.83	9,551.44	✓	✓
CRAFT-b3-D	$\mathtt{m}(4,2,\mathcal{T},\mathtt{cr})$	timeout	**0.05**	**0.05**	timeout	52.42	2.28	✗	✗
LED64-b1-D	$\mathtt{m}(1,1,\mathcal{T},\mathtt{cr})$	**0.93**	1.60	31.90	timeout	5,082.29	timeout	✓	✓
LED64-b1-D	$\mathtt{m}(2,1,\mathcal{T},\mathtt{cr})$	0.96	**0.16**	0.93	timeout	1.04	timeout	✗	✗
LED64-b2-D	$\mathtt{m}(2,1,\mathcal{T},\mathtt{cr})$	6.41	**2.34**	81.85	timeout	4,293.95	timeout	✓	✓
LED64-b2-D	$\mathtt{m}(3,1,\mathcal{T},\mathtt{cr})$	44.55	**0.17**	1.88	timeout	1.60	timeout	✗	✗

ing the bit-flip fault type needs fewer fault type selection inputs than that of set and reset fault types. Note that adding more acceleration techniques does not necessarily make an improvement, e.g., GR·TR$_{sr}$ vs. GR·FE·TR$_{sr}$ on AES-bi-D, because $\sharp\texttt{MaxE}(\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)) = \texttt{n}_e$ and $\sharp\texttt{Clk}(\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)) = \texttt{n}_c$ are encoded as $\texttt{n}_e \leq \sharp\texttt{MaxE}(\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)) \leq \texttt{n}_e$ and $\texttt{n}_c \leq \sharp\texttt{Clk}(\mathsf{V}(\mathcal{S}, \mathbf{B}_\ell, T)) \leq \texttt{n}_c$ before bit-blasting. Remark that FIRMER [37] indeed is CLEAVE when only GR is enabled. Due to space limitations, the results of SMT- and BDD-based verification are reported elsewhere [38], from which the same conclusion can be drawn. Thus, hereafter, we adopt the combination of acceleration techniques GR·TR$_{bf}$.

5.2 Evaluation of Compositional Verification

To evaluate our compositional approach, we compare it with the monolithic one, both of which adopt the combination of acceleration techniques GR·TR$_{bf}$.

The results are reported in Table 3. Overall, our compositional reasoning is very effective, allowing CLEAVE to verify fault-resistance of almost all the benchmarks while their monolithic counterparts often run out of time. For instance, the monolithic BDD-based approach fails to verify all the benchmarks due to the huge number of BDD variables. Indeed, the maximal number of rounds that can be handled is 2 (cf. [38] for details).

In contrast, the compositional reasoning can verify all the benchmarks, except for AES-b2-D and CRAFT-b3-D where even the single-round sub-circuit cannot be verified by the BDD-based approach. For SAT/SMT-based verification, the compositional reasoning takes significantly less time than its monolithic counterpart. Note that the diverse performance between SAT/SMT- and BDD-based approaches is mainly because we use the parallel SAT solver Glucose (8 threads) versus sequential SMT solver bitwuzla, and there is a cost for building (several) BDDs.

6 Related Work

Equivalence and safety checking play an essential role in the design of circuits. Various SAT/SMT-based approaches (e.g., [7,10–12,22]) and BDD-based approaches (e.g., [13,14,17,29]) have been studied. They are orthogonal to our work and cannot be directly applied to check fault-resistance.

Due to the prevalence of fault injection attacks, there are studies for finding the effective fault vectors or checking the effectiveness of the fault vectors provided by users, e.g., [4,23,33]. However, it is virtually impossible to enumerate all the possible fault vectors and valid inputs in practice, thus these approaches are limited in efficiency and scalability. To mitigate these issues, the BDD-based approach, FIVER [31], was proposed which does not need to explicitly enumerate all the possible valid inputs [31], but still has to explicitly enumerate all the possible fault vectors. Very recently, the SAT/SMT-based approach, FIRMER [37], was proposed to implicitly encode all the possible fault vectors into SAT/SMT formulas, and thus no explicit enumeration is required for both possible fault vectors and valid inputs. However, they often fail to verify the entire circuit under all the possible fault vectors and valid inputs. Our compositional approach circumvents the verification of the entire circuit of a large size, and can significantly boost both SAT/SMT-based and BDD-based verification approaches with novel acceleration techniques.

Compositional reasoning is a powerful divide-and-conquer approach for addressing the state-explosion problem. Hence, various compositional reasoning techniques and methods have been investigated, e.g., [19,20,25,27], for safety, equivalence and side-channel security verification. Our compositional reasoning relies on the structural feature of (round-based) cryptographic circuits and the fault-resistance verification problem, thus is different from the prior ones.

Synthesis techniques have been proposed to repair flaws (e.g., [18,32,40]). However, they do not provide security guarantees (e.g., [32,40]) or are limited to one specific type of fault injection attacks (e.g., clock glitch in [18]) and thus may be still vulnerable to other fault injection attacks.

7 Conclusion

We have proposed the first compositional reasoning which decomposes the fault-resistance verification of a whole round-based cryptographic circuit into that of single-round sub-circuits. To efficiently verify single-round sub-circuits, we have proposed various acceleration techniques and studied both SAT/SMT-based and BDD-based approaches. We have implemented our approach in an open-source tool CLEAVE and extensively evaluated it on a set of realistic cryptographic circuits. The experimental results show that our compositional approach and acceleration techniques can significantly improve all the SAT/SMT-based and BDD-based verification approaches, outperforming the state-of-the-art baselines.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aghaie, A., Moradi, A., Rasoolzadeh, S., Shahmirzadi, A.R., Schellenberg, F., Schneider, T.: Impeccable circuits. IEEE Trans. Comput. **69**, 361–376 (2020)
2. Agoyan, M., Dutertre, J., Naccache, D., Robisson, B., Tria, A.: When clocks fail: on critical paths and clock faults. In: Proceedings of the 9th IFIP WG 8.8/11.2 International Conference (CARDIS), pp. 182–193 (2010)
3. Anderson, R.J., Kuhn, M.G.: Low cost attacks on tamper resistant devices. In: Christianson, B., Crispo, B., Lomas, T.M.A., Roe, M. (eds.) Proceedings of the 5th International Workshop on Security Protocols, vol. 1361, pp. 125–136 (1997). https://doi.org/10.1007/BFB0028165

4. Arribas, V., Wegener, F., Moradi, A., Nikova, S.: Cryptographic fault diagnosis using VerFI. In: Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 229–240 (2020)
5. Atzori, L., Iera, A., Morabito, G.: The internet of things: a survey. Comput. Netw. **54**(15), 2787–2805 (2010)
6. Audemard, G., Simon, L.: On the glucose SAT solver. Int. J. Artif. Intell. Tools **27**(1), 1840001:1–1840001:25 (2018)
7. Azarbad, M.R., Alizadeh, B.: Scalable SMT-based equivalence checking of nested loop pipelining in behavioral synthesis. ACM Trans. Design Autom. Electr. Syst. **22**(2), 22:1–22:22 (2017)
8. Baksi, A.: Classical and Physical Security of Symmetric Key Cryptographic Algorithms. Springer, Singapore (2022). https://doi.org/10.1007/978-981-16-6522-6
9. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proc. IEEE **94**(2), 370–382 (2006). https://doi.org/10.1109/JPROC.2005.862424
10. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Proceedings of the 36th Conference on Design Automation (DAC), pp. 317–320 (1999)
11. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), pp. 193–207 (1999)
12. Bruttomesso, R., et al.: A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems. In: Proceedings of the 19th International Conference on Computer Aided Verification (CAV), pp. 547–560 (2007)
13. Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L., Dill, D.L.: Symbolic model checking for sequential circuit verification. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **13**(4), 401–424 (1994)
14. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10ˆ20 states and beyond. In: Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS), pp. 428–439 (1990)
15. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
16. Dehbaoui, A., Dutertre, J., Robisson, B., Tria, A.: Electromagnetic transient faults injection on a hardware and a software implementations of AES. In: Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 7–15 (2012)
17. van Eijk, C.A.J.: Sequential equivalence checking without state space traversal. In: Proceedings of Design, Automation and Test in Europe (DATE), pp. 618–623 (1998)
18. Eldib, H., Wu, M., Wang, C.: Synthesis of fault-attack countermeasures for cryptographic circuits. In: Proceedings of the 28th International Conference on Computer Aided Verification (CAV), pp. 343–363 (2016)
19. Gao, P., Song, F., Chen, T.: Compositional verification of first-order masking countermeasures against power side-channel attacks. ACM Trans. Softw. Eng. Methodol. **33**(3), 79:1–79:38 (2024)
20. Gao, P., Zhang, Y., Song, F., Chen, T., Standaert, F.: Compositional verification of efficient masking countermeasures against side-channel attacks. Proc. ACM Program. Lang. **7**(OOPSLA2), 1817–1847 (2023). https://doi.org/10.1145/3622862
21. Joye, M., Tunstall, M. (eds.) Fault Analysis in Cryptography. Information Security and Cryptography. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29656-7

22. Kaiss, D., Skaba, M., Hanna, Z., Khasidashvili, Z.: Industrial strength SAT-based alignability algorithm for hardware equivalence verification. In: Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design, pp. 20–26 (2007)
23. Khanna, P., Rebeiro, C., Hazra, A.: XfC: a framework for exploitable fault characterization in block ciphers. In: Proceedings of the 54th Annual Design Automation Conference (DAC), pp. 1–6 (2017)
24. Khasidashvili, Z., Skaba, M., Kaiss, D., Hanna, Z.: Theoretical framework for compositional sequential hardware equivalence verification in presence of design constraints. In: Proceedings of the International Conference on Computer-Aided Design, pp. 58–65 (2004)
25. Khasidashvili, Z., Skaba, M., Kaiss, D., Hanna, Z.: Post-reboot equivalence and compositional verification of hardware. In: Proceedings of the 6th International Conference on Formal Methods in Computer-Aided Design, pp. 11–18 (2006)
26. Malkin, T., Standaert, F., Yung, M.: A comparative cost/security analysis of fault attack countermeasures. In: Proceedings of the 3rd International Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 159–172 (2006)
27. McMillan, K.L.: A methodology for hardware verification using compositional model checking. Sci. Comput. Program. **37**(1–3), 279–309 (2000). https://doi.org/10.1016/S0167-6423(99)00030-1
28. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR abs/2006.01621 (2020). https://arxiv.org/abs/2006.01621
29. Pixley, C.: A theory and implementation of sequential hardware equivalence. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **11**(12), 1469–1478 (1992)
30. Richter-Brockmann, J., Sasdrich, P., Güneysu, T.: Revisiting fault adversary models - hardware faults in theory and practice. IEEE Trans. Comput. **72**, 572–585 (2023)
31. Richter-Brockmann, J., Shahmirzadi, A.R., Sasdrich, P., Moradi, A., Güneysu, T.: Fiver - robust verification of countermeasures against fault injections. IACR Trans. Cryptographic Hardware Embed. Syst. **2021**, 447–473 (2021)
32. Roy, I., Rebeiro, C., Hazra, A., Bhunia, S.: SAFARI: automatic synthesis of fault-attack resistant block cipher implementations. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(4), 752–765 (2020)
33. Saha, S., Mukhopadhyay, D., Dasgupta, P.: ExpFault: an automated framework for exploitable fault characterization in block ciphers. IACR Trans. Cryptographic Hardware Embed. Syst. **2018**(2), 242–276 (2018)
34. Selmane, N., Guilley, S., Danger, J.: Practical setup time violation attacks on AES. In: Proceedings of the 7th European Dependable Computing Conference (EDCC), pp. 91–96 (2008)
35. Shahmirzadi, A.R., Rasoolzadeh, S., Moradi, A.: Impeccable circuits II. Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC), pp. 1–6 (2020)
36. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: Proceedings of the 4th International Workshop Redwood Shores on Cryptographic Hardware and Embedded Systems (CHES), pp. 2–12 (2003)
37. Tan, H., Gao, P., Chen, T., Song, F., Wu, Z.: SAT-based formal fault-resistance verification of cryptographic circuits. CoRR abs/2307.00561 (2023)
38. Tan, H., Gao, P., Chen, T., Song, F., Wu, Z.: CLEAVE (2024). https://github.com/S3L-official/CLEAVE
39. Tyagi, A.K., Sreenath, N.: Cyber physical systems: analyses, challenges and possible solutions. Internet Things Cyber-Phys. Syst. **1**, 22–33 (2021)

40. Wang, H., Li, H., Rahman, F., Tehranipoor, M.M., Farahmandi, F.: SoFI: security property-driven vulnerability assessments of ICs against fault-injection attacks. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **41**(3), 452–465 (2021)
41. Zussa, L., Dutertre, J.M., Clediere, J., Tria, A.: Power supply glitch induced faults on fpga: An in-depth analysis of the injection mechanism. In: Proceedings of the IEEE 19th International On-Line Testing Symposium (IOLTS), pp. 110–115 (2013)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Reusable Specification Patterns for Verification of Resilience in Autonomous Hybrid Systems

Julius Adelt[1](✉), Robert Mensing[2], and Paula Herber[1,2]

[1] University of Münster, Münster, Germany
{julius.adelt,paula.herber}@uni-muenster.de
[2] University of Twente, Enschede, The Netherlands
{r.a.mensing,pherber}@utwente.nl

Abstract. Autonomous hybrid systems are systems that combine discrete and continuous behavior with autonomous decision-making, e.g., using reinforcement learning. Such systems are increasingly used in safety-critical applications such as self-driving cars, autonomous robots or water supply systems. Thus, it is crucial to ensure their safety and resilience, i.e., that they function correctly even in the presence of dynamic changes and disruptions. In this paper, we present an approach to obtain formal resilience guarantees for autonomous hybrid systems using the interactive theorem prover KeYmaera X. Our key ideas are threefold: First, we derive a formalization of resilience that is tailored to autonomous hybrid systems. Second, we present reusable patterns for modeling stressors, detecting disruptions, and specifying resilience as a service level response in the differential dynamic logic (d$\mathcal{L}$). Third, we combine these concepts with an existing approach for the safe integration of learning components using hybrid contracts, and extend it towards dynamic adaptations to stressors. By combining reusable patterns for stressors, observers, and adaptation contracts for learning components, we provide a systematic approach for the deductive verification of resilience of autonomous hybrid systems with reduced specification effort. We demonstrate the applicability of our approach with two case studies, an autonomous robot and an intelligent water distribution system.

Keywords: Hybrid Systems · Reinforcement Learning · Formal Methods · Deductive Verification · Resilience · Reusability

1 Introduction

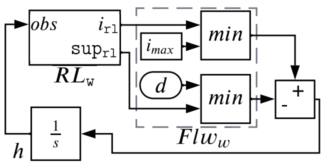
Autonomous hybrid systems (AHS), such as self-driving vehicles, robots, and intelligent water supply systems, combine autonomous decision-making with both discrete and continuous behavior. They often act autonomously in dynamic, safety-critical environments, where failures can cause damage or even endanger

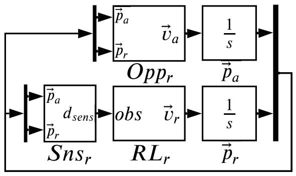
© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 208–228, 2025.
https://doi.org/10.1007/978-3-031-71177-0_14

human lives. As a consequence it is essential to ensure their resilience, i.e., the system's capability to adapt and maintain its correct functioning amidst changes and disruptions. However, the formal verification of AHS poses distinct challenges because of their hybrid nature and the inclusion of learning components like reinforcement learning (RL), which are hard to capture formally. Approaches to overcome this problem using model checking or statistical model checking are impeded by the state-space explosion problem and often only consider resilience up to a certain time bound. Deductive verification, on the other hand, is a powerful approach for scalable mathematical reasoning even for complex unbounded systems, but demands high expertise and manual effort to provide the necessary specifications and invariants. In particular, there is a lack of reusable formal definitions of resilience in the context of AHS and the few existing definitions are not directly applicable for the deductive verification of qualitative resilience guarantees. This is because they are either not formally specified (e.g., [34]), defined for time-bounded quantitative analysis (e.g., [16,17,40]) or tailored to specific classes of systems (e.g., [8]). In this paper, we present a systematic approach for defining, modeling, and verifying resilience of AHS within d$\mathcal{L}$ [55–57] using the interactive theorem prover KeYmaera X [22].

Our approach is based on three key ideas. First, to address the lack of formal qualitative resilience definitions for AHS, we formalize resilience for the deductive verification of AHS based on the informal definition by Laprie [34] by introducing the concept of service levels for AHS which are provided under varying stress conditions. We identify stressors as the key factors causing failures and disruptions, and define service levels to capture dynamic adaptations to stress, such as graceful degradation. Second, to enable systematic deductive verification of resilience properties for AHS, we introduce *stressor patterns* for modeling stressors and *observer patterns* for observing the induced stress. Our stressor patterns facilitate integrating various kinds of stressors in formal AHS models, for example noise, component failures, or unexpected delays. Our observer patterns enable formally capturing the stress induced on a system, and thus to verify the system response as service levels, such as a maximum supply or minimum speed, under varying stress levels. Third, we combine our reusable specification patterns with our own previous approach for deductive verification of Simulink models using d$\mathcal{L}$ [36,37], and for the safe integration of learning into AHS [1,4]. In [1,4], we have specified reusable contract patterns for verifying AHS with RL components with reduced specification and verification effort. In this paper, we extend this approach to reusable *resilience contract patterns*, which link service levels to the stress intensity experienced by the system and ensure that an RL agent dynamically adapts to stressors. By combining reusable patterns for stressors, observers, and dynamic adaptations using service levels, we provide a systematic approach for the deductive verification of resilience of AHS with reduced specification effort.

We demonstrate the applicability of our approach with two case studies: an intelligent water distribution system and an autonomous robot. The for-





$\mathtt{RL_w}$	$\{\text{if}(c \geq T_S)\{c := 0; sup_{rl} := *; i_{rl} := *; ?HC_{\mathtt{w}}; \}$
$\mathtt{Flw_w}$	$i := min(i_{\mathtt{max}}, i_{rl});$
	$d := *; ?d \in [0, sup_{rl}];$
$\mathtt{Plnt_w}$	$\{t' = 1, c' = 1, h' = i - d \,\&\, c \leq T_S\}\}^*$

(a) Intelligent Water Distribution System

$\mathtt{Sns_r}$	$\{d_{sens} := d(\vec{p}_{\mathrm{r}}, \vec{p}_{\mathrm{a}});$
$\mathtt{RL_r}$	$\text{if}(c \geq T_S)\{c := 0; \vec{v}_{\mathrm{r}} := *; ?HC_r; \}$
$\mathtt{Opp_r}$	$\vec{v}_{\mathrm{a}} := *; ?\lvert\vec{v}_{\mathrm{a}}\rvert \leq v_{\mathrm{max,a}};$
$\mathtt{Plnt_r}$	$\{t' = 1, c' = 1, \vec{p}_{\mathrm{r}}' = \vec{v}_{\mathrm{r}}, \vec{p}_{\mathrm{a}}' = \vec{v}_{\mathrm{a}} \,\&\, c \leq T_S\}\}^*$

(b) Autonomous Robot with Opponent

Fig. 1. Simulink (top) and d$\mathcal{L}$ (bottom) models of the Case Studies

mer is based on a model used by MathWorks [42] to demonstrate the RL Toolbox [43], the latter to demonstrate the Robotics System Toolbox [44].

The rest of this paper is structured as follows: we introduce preliminaries in Sect. 2 and our approach in Sect. 3. We present verification results in Sect. 4, discuss related work in Sect. 5, and conclude in Sect. 6.

2 Preliminaries and Case Studies

In this section, we use our two case studies to introduce Simulink and the RL Toolbox, d$\mathcal{L}$, Simulink2d$\mathcal{L}$, and our approach for safe RL using contracts.

2.1 Case Studies in Simulink

Simulink [45] is an industrially well established graphical modeling language for AHS. Simulink is block based and provides a large selection of predefined blocks with discrete or continuous behavior, which can be connected via signals. The semantics of Simulink is informally defined in [45]. The RL Toolbox enables directly integrating and simulating RL agents in Simulink models via an RL agent block, which executes an RL algorithm at discrete time steps.

The upper part of Fig. 1a shows a Simulink model of a reservoir of an intelligent water distribution system (IWDS) based on [47]. In [3], we have presented an approach to safely optimize a similar system using a combination of deductive verification and a statistical model checking based learning approach. Using this approach, an RL agent successfully optimizes the supply provided by the system with a given energy budget by decreasing the inflow (e.g., by switching off pumps) whenever the demand is low and by reducing the maximum available supply if necessary due to pump failures. The Simulink model has a constant maximum inflow rate $i_{\mathtt{max}}$. The RL agent ($\mathtt{RL_w}$) can choose a reduced inflow i_{rl} and a maximum supply sup_{rl}, which sets a limit on the actual demand d. The reservoir water level h evolves by integrating the difference of the inflow $\min(i_{\mathtt{max}}, i_{rl})$ and demand $min(d, sup_{rl})$, which are computed in the $\mathtt{Flw_w}$ subsystem.

The upper part of Fig. 1b shows a Simulink model of an autonomous robot in a factory inspired by [41]. The autonomous robot is dynamically assigned goals within a factory, and its RL controller tries to get the robot there as fast as possible without colliding with moving opponents. We have demonstrated that the robot reaches goals safely for a similar system in [4]. The Simulink model consists of an RL controller ($\texttt{RL}_\texttt{r}$), which receives distance data from a sensor ($\texttt{Sns}_\texttt{r}$), and a second controller for the opponent ($\texttt{Opp}_\texttt{r}$). The RL agent can choose the velocity and direction of the RL robot ($\vec{v}_\texttt{r}$). The positions of $\vec{p}_\texttt{r}$ and $\vec{p}_\texttt{a}$ evolve continuously with axial velocities ($\vec{v}_\texttt{r}$ and $\vec{v}_\texttt{a}$) respectively.

2.2 Differential Dynamic Logic d$\mathcal{L}$ and Simulink2d$\mathcal{L}$

Differential dynamic logic (d$\mathcal{L}$) [55–57] is a logic for formally specifying and reasoning about properties of hybrid systems, which are modeled as hybrid programs (HP). Hybrid Programs are build from the following syntax: $\alpha;\beta$ is a sequential composition of two HP α and β. α^* is a non-deterministic repetition. $x := e$ is a discrete assignment of term e to variable x. $x := *$ assigns a non-deterministically chosen value to x. $\alpha ++ \beta$ is a non-deterministic choice. $?Q$ is a test formula. $if(Q)\{\alpha\}else\{\beta\}$ is syntactic sugar for $\{?Q;\alpha ++ ?\neg Q;\beta\}$. $\{x_1' = \eta_1, \ldots, x_n' = \eta_n \;\&\; Q\}$ is a continuous evolution, where variables x_i evolve with differential equations $x_i' = \eta_i$ while an evolution domain Q is satisfied. Furthermore, in this paper we use $x \in [l, u]$ as syntactic sugar for $l \leq x \wedge x \leq u$. d$\mathcal{L}$ provides modalities $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ for reasoning about reachable states. Safety specifications are expressed as $pre \rightarrow [\alpha]post$ and can be verified using KeYmaera X [22]. Proofs in KeYmaera X are based on the d$\mathcal{L}$ sequent calculus.

In [36], we have presented an automated transformation from Simulink into d$\mathcal{L}$, called Simulink2d$\mathcal{L}$. This provides us with a formal representation of a given Simulink model, and thus enables formal verification of Simulink models using KeYmaera X. Furthermore, we have defined the concept of *hybrid contracts* (HC) for compositional verification of Simulink models in [37]. HC can be defined for components of Simulink models as d$\mathcal{L}$ formulas with $hc = (\phi_{in}, \phi_{out})$, where ϕ_{in} are input assumptions and ϕ_{out} are output and trajectory guarantees. HC can be verified for components individually and replace these components during transformation. To integrate RL agents safely into the transformation, the safe behavior of RL agents can also be defined using HC [4]. HC can be used as shields [23] during simulation to enforce safe behavior of RL agents.

The lower part of Fig. 1a shows a d$\mathcal{L}$ model of the IWDS. The RL agent ($\texttt{RL}_\texttt{w}$) is captured by a conditional hybrid program. If the sample time elapses $c \geq T_S$ the agent selects safe actions i_{rl} and sup_{rl} non-deterministically but in compliance to $HC_\texttt{w}$. $\texttt{Flw}_\texttt{w}$ computes the current inflow i and demand d. Continuous behavior is captured in the continuous evolution ($\texttt{Plnt}_\texttt{w}$). The water level h evolves with $h' = i - d$, the clock c and simulation time t evolve with constant rate 1. The evolution domain $c \leq T_S$ ensures that no sample times of the RL agent are missed. The global simulation loop is modeled by a nondeterministic repetition.

The lower part of Fig. 1b shows a d$\mathcal{L}$ model of the autonomous robot. The sensor ($\texttt{Sns}_\texttt{r}$) assigns the distance $d(\vec{p}_\texttt{r}, \vec{p}_\texttt{a},)$ to a variable d_{sens}. The RL controller ($\mathtt{RL_r}$) chooses new velocities $\vec{v}_\mathtt{r}$ according to its hybrid contract $HC_\mathtt{r}$. The opponent $\mathtt{Opp_r}$ chooses velocities $\vec{v}_\mathtt{a}$ limited by $?|\vec{v}_\mathtt{a}| \leq v_\mathtt{max,a};$. In the continuous evolution, the positions $\vec{p}_\mathtt{a}$, $\vec{p}_\mathtt{r}$, the sampling time clock c and simulation time t evolve within the domain constraint $c \leq T_S$. We use $\vec{v}$ as an abbreviation for axial velocities $(\vec{v_x}, \vec{v_y})$ and $\vec{p}$ for coordinates $(\vec{x}, \vec{y})$.

Table 1. Threshold Pattern [1] and derived Contracts for IWDS and Robot

	Property	Hybrid Contract
Threshold Pattern	$pre \rightarrow [\alpha] \quad var_{sc} \sim \theta$	$var_{sc} + wcr(state, action, T_S) \sim \theta$
IWDS HC_w	$pre \rightarrow [\alpha] \quad h \geq h_{min}$	$h + (i_{rl} - sup_{rl}) \cdot T_S \geq h_{min}$
Robot HC_r	$pre \rightarrow [\alpha]\, d(\vec{p}_\mathtt{r}, \vec{p}_\mathtt{a}) > \theta_\mathtt{evd}$	$d_{sens} - (\lvert\vec{v}_\mathtt{r}\rvert + v_\mathtt{max,a}) \cdot T_S > \theta_\mathtt{evd}$

2.3 Reusable Contracts for Safe Integration of Learning

In [1], we have introduced reusable HC patterns for addressing common verification challenges in AHS. These patterns are derived from recurring elements in AHS verification problems and provide templates for the specification of contracts and invariants for learning components. As an example, Table 1 shows the *threshold pattern* and its application to our two case studies. The pattern specifies the contract that is needed to ensure that the variable var_{sc} stays within a given threshold θ ($pre \rightarrow [\alpha]\, var_{sc} \sim \theta$ with $\sim \in \{<, \leq, =, \geq, >\}$). To ensure this property on system level, the RL agent has to maintain the threshold within the next sample time while accounting for the systems worst case reaction (wcr) to the current $state$, $action$ and the sample time (T_S) of the RL agent.

In the IWDS, the RL agent has to keep the water level above a minimum $h \geq h_{min}$. As an action, the agent may choose an inflow i_{rl} and limit the outflow to a maximum supply sup_{rl}. The worst case reaction of the environment is a demand that fully exploits the supply limit ($d = sup_{rl}$). For the robot, a crucial safety requirement is to maintain a minimum distance $d(\vec{p}_\mathtt{r}, \vec{p}_\mathtt{a}) > \theta_\mathtt{evd}$ to the opponent or to stop if the opponent further decreases the distance. The RL agents threshold contract ensures that the chosen velocity $\vec{v}_\mathtt{r}$ maintains the distance $\theta_\mathtt{evd}$ from the opponents current position. To stop if $\theta_\mathtt{evd}$ can no longer be maintained, we add a disjunction to the contract ($HC_r \vee |\vec{v}_\mathtt{r}| = 0$) (not shown in the table).

3 Reusable Patterns for Deductive Verification of Resilience in Autonomous Hybrid Systems

Autonomous hybrid systems (AHS) may face various stressors, for example, sensor noise, component failures, or unexpected delays. It is highly desirable to ensure that AHS are resilient, i.e., that they still function correctly in the presence of such stressors. There exist various definitions of resilience [8,16,17,34,40]. However, there is a lack of reusable formal definitions of resilience for AHS specifically, especially for the deductive verification of qualitative resilience guarantees.

In this paper, we follow the informal definition provided by Laprie in [34]: *"The persistence of service delivery that can justifiably be trusted, when facing changes"*. From this definition, we derive a formalization of resilience for the deductive verification of AHS via reusable specification patterns using stressors to describe (safety-critical) changes, and service levels to describe service delivery.

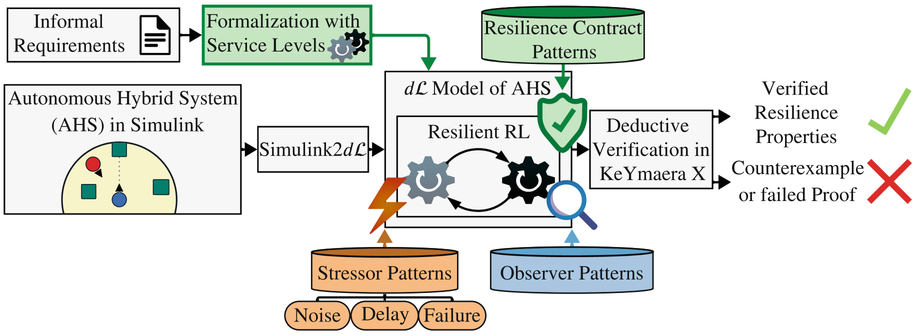

Informal Requirements
Formalization with Service Levels
Resilience Contract Patterns
Autonomous Hybrid System (AHS) in Simulink
Simulink2$d\mathcal{L}$
$d\mathcal{L}$ Model of AHS
Resilient RL
Deductive Verification in KeYmaera X
Verified Resilience Properties
Counterexample or failed Proof
Stressor Patterns
Noise
Delay
Failure
Observer Patterns

Fig. 2. Our Approach for Deductive Verification of Resilience in AHS

Our overall approach is shown in Fig. 2. Our process starts with an *Autonomous Hybrid System (AHS)* modeled in Simulink, which includes a reinforcement learning (RL) agent for autonomous decision-making, and *Informal Requirements*, including resilience. The AHS is transformed into a *d$\mathcal{L}$ Model* using the *Simulink2d$\mathcal{L}$* transformation [4,36]. To establish a structured approach for formalizing and verifying resilience properties, we introduce *Service Levels* to formalize the system's adaptive response to stressors. This means, for example, that we describe graceful degradation using a degraded service level together with safety thresholds that are still maintained under stress.

For verifying resilience in AHS, we need to formally model stressors. However, this typically requires high expertise. In particular, it is often unclear how to specify changes in behavior and the intensity of stress induced by given stressors in a formally modeled system. To tackle this, we introduce reusable *Stressor Patterns*. They are designed to capture various disturbances and changes, ranging from discrete or continuous *noise* over timed *delays* to complete *failures*. In our definition, stressors strictly extend the possible behavior of components with non-determinism. This facilitates easy integration of stressors into existing d$\mathcal{L}$ models. Furthermore we avoid the need to provide probability distributions, which are often not available or hard to obtain.

To deductively verify and safely integrate learning in AHS modeled in Simulink, we have proposed an approach to replace RL components by hybrid contracts that describe safe actions in [4]. These contracts can be used as shields via automatically generated runtime monitors [23,49]. In [1] we have proposed reusable contract patterns for common verification problems in AHS. In this paper, to ensure *Resilient RL* for AHS, we extend this approach with reusable *Resilience Contract Patterns*, which link appropriate service levels to the stress intensity experienced by the system. With such *Resilience Contracts*, we can enforce that the RL agent dynamically adapts to stressors and disruptions using service levels.

To be able to verify an overall AHS under different stress levels, capturing the stress intensity induced by stressors formally is desirable. To address this, we propose reusable *Observer Patterns*. In our definition, observers may never change the system's behavior but are used to passively capture the dynamic effects of stressors and stress intensity on the system.

Table 2. Service Levels and corresponding Safety Thresholds

Case Study	Service Level	enabled Actions A_i	Threshold under low stress ($\theta_{\mathtt{ls}}$)	Threshold under high stress ($\theta_{\mathtt{hs}}$)
IWDS	`IWDSFull`	$i_{rl} \in [0, i_{\mathtt{max}}], sup_{rl} = sup_{\mathtt{max}}$	$h \geq h_{\mathtt{max}}$	$h \geq h_{\mathtt{dgr}}$
	`IWDSDeg`	$i_{rl} \in [0, i_{\mathtt{max}}], sup_{rl} = sup_{\mathtt{dgr}}$	$h \geq h_{\mathtt{dgr}}$	$h \geq h_{min}$
	`IWDSNo`	$i_{rl} \in [0, i_{\mathtt{max}}], sup_{rl} = 0$	$h \geq h_{min}$	$h \geq h_{min}$
Rob	`RobEvd`	$\vec{v}_{\mathtt{r}} \in [v_{\mathtt{min,r}}, v_{\mathtt{max,r}}]$	$d(\vec{p}_{\mathtt{r}}, \vec{p}_{\mathtt{a}}) > \theta_{\mathtt{evd}}$	$d(\vec{p}_{\mathtt{r}}, \vec{p}_{\mathtt{a}}) > \theta_{\mathtt{stp}}$
	`RobStop`	$\vec{v}_{\mathtt{r}} = 0$		

With our reusable stressor, resilience contracts, and observer patterns, we enable formal specifications of resilient systems in d$\mathcal{L}$, and their deductive verification using KeYmaera X [22].

In the following subsections, we introduce the concept of service levels as means for dynamic adaptation, our reusable patterns for stressors and observers, as well as resilience contracts, in more detail.

3.1 Formalization of Resilience Using Service Levels

In AHS, learning components dynamically adapt to changes in the environment. To ensure safety and resilience, we have to make sure that the system remains operational in the presence of stressors. To achieve this, we want to verify that the system satisfies requirements under varying stress levels for all possible adaptations. However, the number of possible adaptations is potentially infinite.

To overcome this problem, we introduce the idea of (a finite number of) service levels (e.g., full, degraded, and no service) to define resilience properties. For each service level, we define ranges of actions (e.g., inflow and supply or speed in our case studies) together with safety thresholds, which can be guaranteed at each service level under varying stress conditions. We can use this to describe high service levels in the absence of stress, and graceful degradation under stress by defining thresholds where we degrade to lower service. Note that within each service level, the system may still choose arbitrary actions from the given range, which enables, for example, learning components or RL agents to safely optimize w.r.t performance properties while resilience guarantees are maintained.

Table 2 shows service levels together with the enabled actions A_i and corresponding safety thresholds under high or low stress for the IWDS and the robot. For the IWDS, at full service `IWDSFull`, the highest possible supply $sup_{\mathtt{max}}$ is enabled. In the absence of high stress, a water level $h \geq h_{\mathtt{max}}$ can be maintained. If high stress occurs, $h \geq h_{\mathtt{dgr}}$ with $h_{\mathtt{dgr}} < h_{\mathtt{max}}$ must still be maintained to ensure that we can gracefully degrade to the lower service level `IWDSDeg`. If stressors persist, the system degrades to `IWDSDeg`, where only a degraded water level $h \geq h_{\mathtt{dgr}}$ is maintained even in the absence of high stress. Under high stress, the minimum water level $h_{min} < h_{\mathtt{dgr}}$ has to be maintained. If these thresholds can be no longer maintained, the system degrades even further to `IWDSNo`, where no supply is provided and only h_{min} is guaranteed at all stress levels.

Table 3. Stressor Patterns

Discrete Noise (with $\circ \in \{\pm, \cdot\}$)			
α	$x := \xi;$	$S(\alpha)$	$\eta := *; ?\eta \in [\eta_{\mathtt{min}}, \eta_{\mathtt{max}}]; x := \xi \circ \eta;$
$\mathtt{Sns_r}$	$d_{sens} := d(\vec{p}_{\mathtt{r}}, \vec{p}_{\mathtt{a}});$	$\mathtt{Sns_r}_{\eta}$	$\eta := *;\ ?\eta \leq \eta_{\mathtt{high}}; d_{sens} := d(\vec{p}_{\mathtt{r}}, \vec{p}_{\mathtt{a}}) + \eta;$
Continuous Noise (with $\circ \in \{\pm, \cdot\}$)			
α	$\{..., x' = \xi\}$	$S(\alpha)$	$\eta := *; ?\eta \in [\eta_{\mathtt{min}}, \eta_{\mathtt{max}}]; \{..., x' = \xi \circ \eta\}$
$\mathtt{Plnt_w}$	$\{\ldots h' = i - d \,\&\, c \leq T_S\}$	$\mathtt{Plnt_{w}}_{\mathtt{l}}$	$\mathtt{l} := *; ?\mathtt{l} \in [0, \mathtt{l}_{\mathtt{high}}];$ $\{\ldots h' = i - d - \mathtt{l} \,\&\, c \leq T_S\}$
$\mathtt{Plnt_r}$	$\{\ldots \vec{p}_{\mathtt{r}}' = \vec{v}_{\mathtt{r}}, \vec{p}_{\mathtt{a}}' = \vec{v}_{\mathtt{r}} \,\&\, c \leq T_S\}$	$\mathtt{Plnt_r}_{\delta}$	$\delta := *; ?\delta \in [-\delta_{\mathtt{high}}, \delta_{\mathtt{high}}];$ $\{\ldots \vec{p}_{\mathtt{r}}' = \vec{v}_{\mathtt{r}} \cdot \delta, \vec{p}_{\mathtt{a}}' = \vec{v}_{\mathtt{r}} \,\&\, c \leq T_S\}$
Failure			
α	α	$S(\alpha)$	$\{\alpha + + \alpha_{\mathtt{f}}\}$
$\mathtt{Flw_w}$	$i := min(i_{\mathtt{max}}, i_{rl});$	$\mathtt{Flw_{wf}}$	$\{i := min(i_{\mathtt{max}}, i_{rl}); + + i := 0;\}$
Delay			
α	$\text{if}(c \geq T_S)\{c := 0; \ldots\}\ldots$ $\{c' = 1, \ldots \,\&\, (c \leq t_s)\}$	$S(\alpha)$	$t_{\Delta} := *;\ ?t_{\Delta} \in [t_{\Delta\mathtt{min}}, t_{\Delta\mathtt{max}}];$ $\text{if}(c \geq T_S + t_{\Delta})\{c := 0; \ldots\}$ $\{c' = 1, \ldots \,\&\, (c \leq t_s + t_{\Delta})\}$
$\mathtt{RL_w}$	$\text{if}(c \geq T_S)$ $\{c := 0; sup_{rl} := *; i_{rl} := *; ?HC_{\mathtt{w}}; \}\ldots$	$\mathtt{RL_w}_{t_{\Delta}}$	$t_{\Delta} := *; ?t_{\Delta} \in [0, t_{\Delta,\mathtt{high}}]; \text{if}(c \geq T_S + t_{\Delta})$ $\{c := 0; sup_{rl} := *; i_{rl} := *; ?HC_{\mathtt{w}}; \}\ldots$
$\mathtt{RL_r}$	$\text{if}(c \geq T_S)\{c := 0; \vec{v}_{\mathtt{r}} := *; ?HC_r; \}\ldots$	$\mathtt{RL_r}_{t_{\Delta}}$	$t_{\Delta} := *; ?t_{\Delta} \in [0, t_{\Delta,\mathtt{high}}];$ $\text{if}(c \geq T_S + t_{\Delta})\{c := 0; \vec{v}_{\mathtt{r}} := *; ?HC_r; \}\ldots$

For the robot, at full service level `RobEvd`, the robot is moving and any speed within $[v_{\mathtt{min,r}}, v_{\mathtt{max,r}}]$ may be chosen. In the absence of high stress, the robot maintains an evasion distance $\theta_{\texttt{evd}}$, where opponents have room to safely evade. If stress occurs, the robot at least maintains a stopping distance $\theta_{\texttt{stp}}$, where it can still safely stop before a potential collision. If these thresholds can not be maintained, the robot stops, i.e., degrades to `RobStop`.

3.2 Reusable Stressor Patterns

The inherent uncertainties and dynamic nature of stressors present a significant challenge and their formal specification requires high expertise and manual effort. To address this problem, we introduce *reusable stressor patterns.* These patterns can be used to formally define the effect of changes and disturbances such as noisy sensors, component failures, or unexpected delays. In our reusable stressor patterns, we over-approximate possible changes with non-determinism. With that, we deliberately avoid the need to provide probability distributions, which are often not available or hard to obtain due to the unpredictable nature of stress factors. Our stressor patterns strictly extend the possible behavior of HPs, thus all behavior of the original HS without stressors is still part of the reachable states. We propose four types of stressor patterns for modeling typical stressors in AHS: *discrete* and *continuous noise*, execution *delays*, and *failures.* The patterns and their application to our case studies are illustrated in Table 3.

The *Discrete Noise* pattern models random or unwanted signals. It broadens the range of possible assignments to the variable of a discrete signal x by adding a non-deterministically chosen noise value η. η can be limited using a Test $?\eta \in [\eta_{\texttt{min}}, \eta_{\texttt{max}}];$. $\texttt{Sns}_{\texttt{r}\eta}$ illustrates the application of this pattern with the robot sensor. To ensure that an added stressor variable does not exclude runs of the original HP, the range of possible values must contain the identity element for the operator $\circ$, i.e., 0 for additive ($\circ = \pm$) and 1 for multiplicative noise ($\circ = \cdot$).

Continuous Noise can influence the continuous behavior of components. For example, we can have a motion drift, where a robot's actual trajectory deviates from its intended path over time, caused by factors such as wheel slippage or actuator inaccuracies. To model continuous noise, our pattern adds a disturbance value η to the derivative of x. We illustrate this pattern with a continuous leakage of the plant of the IWDS ($\texttt{Plnt}_{\texttt{wl}}$) and a motion drift of the robot ($\texttt{Plnt}_{\texttt{r}\delta}$).

The *Failure* pattern models failures using a non-deterministic choice between an original HP α and a failure HP $\alpha_{\texttt{f}}$, which models the behavior of α under failure. By retaining the original HP α as one of the choices, the original runs of the model are preserved and we can reason over arbitrary alternations α and α_{fail}. We illustrate this pattern with the IWDS ($\texttt{Flw}_{\texttt{wf}}$). The failure model introduces pump failures by setting the inflow rate to zero ($i := 0;$).

The *Delay* pattern introduces variability to the execution time of discrete components, such as an RL agent, by adding a non-deterministic delay (t_Δ) into their periodicity. The sampling clock (c) is then permitted to exceed its normal cycle (T_S) by t_Δ in all tests and evolution domains. We illustrate this pattern with the RL agent of the IWDS ($\texttt{RL}_{\texttt{w}t_\Delta}$) and robot ($\texttt{RL}_{\texttt{r}t_\Delta}$). Note that we omit the evolution domain in the delay examples for brevity.

3.3 Safe Integration of Learning Using Resilience Contract Patterns

To ensure that a learning component adapts correctly and safely switches between service levels as defined above, we adopt our approach for the safe integration of learning presented in [1,4]. There, we have defined safe actions for learning components using reusable contract patterns to address recurring verification problems in AHS. To exploit this concept for the formal verification of resilience and dynamic adaptations using service levels, we introduce reusable hybrid contract patterns for resilience via dynamic adaptation to stressors.

Table 4. Resilience Contract Pattern and Service Recovery

Resilience	$a \in A$	$\wedge \quad var_{sc} + wcr(s, a, t) \geq \theta_{\mathtt{ls}}$
$HC_{\mathtt{IWDSFull}}$	$i_{rl} \in [0, i_{\mathtt{max}}] \wedge sup_{rl} = sup_{\mathtt{max}}$	$\wedge\, h - wcr_{\mathtt{ls}}(s, (i_{rl}, sup_{rl}), T_S) \geq h_{\mathtt{max}}$ $\wedge\, h - wcr_{\mathtt{hs}}(s, (i_{rl}, sup_{rl}), T_S) \geq h_{\mathtt{dgr}}$
$HC_{\mathtt{IWDSDeg}}$	$i_{rl} \in [0, i_{\mathtt{max}}] \wedge sup_{rl} = sup_{\mathtt{dgr}}$	$\wedge\, h - wcr_{\mathtt{ls}}(s, (i_{rl}, sup_{rl}), T_S) \geq h_{\mathtt{dgr}}$ $\wedge\, h - wcr_{\mathtt{hs}}(s, (i_{rl}, sup_{rl}), T_S) \geq h_{min}$
$HC_{\mathtt{IWDSDeg}}$	$i_{rl} \in [0, i_{\mathtt{max}}] \wedge sup_{rl} = 0$	$\wedge\, h - wcr_{\mathtt{ls}}(s, (i_{rl}, sup_{rl}), T_S) \geq h_{min}$ $\wedge\, h - wcr_{\mathtt{hs}}(s, (i_{rl}, sup_{rl}), T_S) \geq h_{min}$
$HC_{\mathtt{RobEvd}}$	$\vec{v}_{\mathtt{r}} \in [v_{\mathtt{min,r}}, v_{\mathtt{max,r}}]$	$\wedge \quad d_{sens} - wcr_{\mathtt{ls}}(s, \vec{v}_{\mathtt{r}}, T_S) > \theta_{\mathtt{evd}}$ $\wedge \quad d_{sens} - wcr_{\mathtt{hs}}(s, \vec{v}_{\mathtt{r}}, T_S) > \theta_{\mathtt{stp}}$
$HC_{\mathtt{RobStop}}$	$\vec{v}_{\mathtt{r}} = 0$	
Service Recovery		$var_{sc} + wcr_{\mathtt{ls}}(s, a, t) \geq var_{sc} + \mathtt{rr} \cdot t$
$HC_{\mathtt{IWDSRec}}$		$h + wcr_{\mathtt{ls}}(s, (i_{rl}, sup_{rl}), T_S) \geq \quad h + \mathtt{rr} \cdot T_S$

Table 5. Worst Case Reactions under Low and High Stress for the two Case Studies

Case Study	Stressor	Low Stress ($wcr_{\mathtt{ls}}$)	High Stress ($wcr_{\mathtt{hs}}$)
IWDS	fail	$+(i_{rl} - sup_{rl}) \cdot T_S$	$+(0 - sup_{rl}) \cdot T_S$
	delay	$+(i_{rl} - sup_{rl}) \cdot (T_S + t_{\Delta,\mathtt{low}})$	$+(i_{rl} - sup_{rl}) \cdot (T_S + t_{\Delta,\mathtt{high}})$
	leak	$+(i_{rl} - sup_{rl} - \mathtt{l}_{\mathtt{low}}) \cdot T_S$	$+(i_{rl} - sup_{rl} - \mathtt{l}_{\mathtt{high}}) \cdot T_S$
Robot	noise	$-(\lvert\vec{v}_{\mathtt{r}}\rvert + v_{\mathtt{max,a}}) \cdot T_S - \eta_{\mathtt{low}}$	$-(\lvert\vec{v}_{\mathtt{r}}\rvert + v_{\mathtt{max,a}}) \cdot T_S - \eta_{\mathtt{high}}$
	delay	$-(\lvert\vec{v}_{\mathtt{r}}\rvert + v_{\mathtt{max,a}}) \cdot (T_S + t_{\Delta,\mathtt{low}})$	$-(\lvert\vec{v}_{\mathtt{r}}\rvert + v_{\mathtt{max,a}}) \cdot (T_S + t_{\Delta,\mathtt{high}})$
	drift	$-(\lvert\vec{v}_{\mathtt{r}} \cdot \delta_{\mathtt{low}}\rvert + v_{\mathtt{max,a}}) \cdot T_S$	$-(\lvert\vec{v}_{\mathtt{r}} \cdot \delta_{\mathtt{high}}\rvert + v_{\mathtt{max,a}}) \cdot (T_S)$

Resilience Contract Patterns. The top row of Table 4 presents the pattern we use to define resilience contracts for learning components within a given AHS. A primary challenge in defining contracts for learning components in AHS is that these components typically select actions at discrete sample times, while thresholds must be maintained throughout all continuous evolutions. As detailed in Sect. 2.3, an RL agent, for example, must maintain safety thresholds within the next sample time while accounting for the system's worst-case reaction (wcr) relative to the current state (s), action (a), and the sample time (T_S) of the RL agent. To ensure resilience, we utilize a conjunction of two threshold patterns: one for maintaining the threshold under low stress $\theta_{\mathtt{ls}}$ with a worst-case reaction in the absence of high stress $wcr_{\mathtt{ls}}$, and another for maintaining the threshold under high stress $\theta_{\mathtt{ls}}$ with a worst-case reaction in the presence of high stress $wcr_{\mathtt{hs}}$. This ensures that while providing a service level, the system can respond to both low stress scenarios and high stress conditions by maintaining the corresponding thresholds. The contracts for the IWDS ($HC_{\mathtt{IWDSFull}}, HC_{\mathtt{IWDSDeg}}, HC_{\mathtt{IWDSNo}}$) and autonomous robot ($HC_{\mathtt{RobEvd}}, HC_{\mathtt{RobStop}}$) in Table 4 utilize this pattern with the action ranges and corresponding thresholds from Table 2. In all of these definitions, the worst case reaction of the environment depends on the stress level.

Worst Case Reactions Under Stress. Table 5 shows the definitions of the worst case reactions $wcr_{\mathtt{ls}}$ and $wcr_{\mathtt{hs}}$ for our case studies for various stressors under low (ls) and high stress (hs). For pump failures in the IWDS case study (stressor *fail*), low stress means that no pump fails. The worst case reaction of the environment is then that the current water level is increased by the inflow chosen by the RL agent i_{rl}, while it is decreased by the full maximum supply sup_{rl} as the demand fully exploits the available supply within the next sample time T_S. In case of high stress, i.e., if the pump fails, the inflow becomes 0, and the current water level is decreased by sup_{rl} only within the next sample time. If the execution of the learning agent is delayed (stressor *delay*), the time for which the worst case reaction is considered is increased by $t_{\Delta,\mathtt{low}}$ under low stress resp. $t_{\Delta,\mathtt{high}}$ under high stress. If the water tank is leaking (stressor *leak*), the IWDS looses water at rate $\mathtt{l}_{\mathtt{low}}$ under low stress and at rate $\mathtt{l}_{\mathtt{high}}$ under high stress. For the robot case study with added sensor noise (stressor *noise*), the worst case reaction of the environment is that the distance to the opponents is reduced by the chosen speed of the robot $\vec{v}_{\mathtt{r}}$, the maximum speed of the opponent $v_{\mathtt{max,a}}$, and the real distance is additionally reduced by the measurement error $\eta_{\mathtt{low}}$ under low stress resp. $\eta_{\mathtt{high}}$ under high stress. If the execution of the learning agent is delayed (stressor *delay*) in the robot case study, the time for which the worst case reaction is considered is again increased by $t_{\Delta,\mathtt{low}}$ resp. $t_{\Delta,\mathtt{high}}$. If the robot is

Table 6. Embedding for Adaptation via Service Level Contracts

Compositional Embedding	IWDS	Robot
if $(c \geq T_S)$ { $c := 0$; $a := *$;	$i_{rl} := *; sup_{rl} := *$;	$\vec{v}_{\mathtt{r}} := *$
if $(\exists action : HC_{\mathtt{max}}(s, action))$ { $?HC_{\mathtt{max}}(s, a)$; }	$HC_{\mathtt{IWDSFull}}$	$HC_{\mathtt{RobEvd}}$
elseif $(\exists action : HC_{\mathtt{max-1}}(s, action))$ { $?HC_{\mathtt{max-1}}(s, a)$; }	$HC_{\mathtt{IWDSDeg}} \wedge HC_{\mathtt{IWDSRec}}$	$HC_{\mathtt{RobStop}}$
elseif $(\exists action : HC_{\mathtt{max-2}}(s, action))$ ... }	$HC_{\mathtt{IWDSNo}} \wedge HC_{\mathtt{IWDSRec}}$	

drifting (stressor *drift*), the axial velocity chosen by the RL agent is in the worst case increased by a factor $\delta_{\mathtt{low}}$ under low stress resp. $\delta_{\mathtt{high}}$ under high stress.

Safe Recovery to higher Service Levels. Our resilience contracts ensure graceful degradation, i.e., that appropriate service levels are chosen and associated thresholds are maintained. However, these contracts do not guarantee that the system will automatically recover to a higher service level during periods of reduced stress. Our service recovery pattern in the last two rows of Table 4 captures this by ensuring that, under low stress ($wcr_{\mathtt{ls}}$), the safety critical variable increases by at least the recovery rate $\mathtt{rr}$ at each sampling step. With $\mathtt{rr} > 0$, the system reaches a higher service level eventually. $HC_{\mathtt{IWDSRec}}$ applies this pattern to the IWDS. Note that we can't ensure recovery for the robot, as even under low stress, the robot might be forced to stop infinitely by the opponent.

Dynamic Adaptation Using Resilience Contracts. Our resilience contracts shown in Table 4 and 5 ensure that RL agents may only choose actions that are resilient in the sense that stress-dependent safety thresholds are maintained on each service level. To integrate these contracts into a d$\mathcal{L}$ model of the overall AHS, we use a compositional embedding as shown in Table 6, which can be used for an arbitrary but fixed number of service levels. As described in Sect. 2.2, we embed RL agents into a given d$\mathcal{L}$ model via a conditional discrete HP [4], where a new action that satisfies the HC is non-deterministically chosen at each sample time ($c \geq T_S$). To facilitate dynamic adaptation using resilience contracts, we additionally propose a hierarchical *if-else* contract composition that enforces the highest possible service level. This structure sequentially evaluates the applicability of service levels from higher to lower, starting at service level i. If an action exists that fulfills the service level, the non-deterministically chosen action of the RL agent is constrained to the respective service level contract by a test. Otherwise, we check the next lower service level for applicability. The second and third columns of Table 6 show the actions and the hierarchy of service levels for both the IWDS and for the autonomous robot. The highest service level for the IWDS is `IWDSFull` and `RobEvd` for the robot. In case of the IWDS, $HC_{\mathtt{IWDSRec}}$ additionally ensures service recovery in case of degraded or no service.

3.4 Reusable Observer Patterns

So far, we have introduced stressors patterns that introduce stress into a given AHS and resilience contracts with a compositional embedding to dynamically adapt to stress by switching between service levels. We now introduce observer patterns to systematically track the current stress and the system state when disruptions occur. This enables us to *observe* the stress imposed on a given AHS and to relate it to appropriate service levels as a system response. With this, we define resilience specifications which can then be verified using KeYmaera X.

Table 7 shows our observer patterns and applications to the IWDS and robot case studies. The upper part of Table 7 shows our observer pattern for failures. We introduce two observer variables, $\mathtt{f}$ and $\mathtt{f_{old}}$. If a failure occurs, i.e., $\alpha_{\mathtt{f}}$ is

Table 7. Observer Patterns

Observer Pattern		$\mathtt{f}_{\mathtt{old}} := \mathtt{f}; \{\mathtt{f} := 0; \alpha + + \mathtt{f} := 1; \alpha_{\mathtt{fail}}\}; \text{if}(cond(\mathtt{f}, \mathtt{f}_{old}))\{store(v_1, ...v_n)\}$
IWDS (fail)	$\mathtt{Flw}_{\mathtt{wf}\circ}$	$\mathtt{f}_{\mathtt{old}} := \mathtt{f}; \{\mathtt{f} := 0; i := min(i_{\mathtt{max}}, i_{rl}); + + \mathtt{f} := 1; i := 0;\}$
		$\text{if}(\mathtt{f} > \mathtt{f}_{\mathtt{old}})\{t_{\mathtt{fail}} := t; h_{\mathtt{fail}} := h; sup_{rl\,\mathtt{fail}} := sup_{rl}; \}$
		$\text{if}(\mathtt{f} < \mathtt{f}_{\mathtt{old}})\{t_{\mathtt{repair}} := t; \}$
Observer Pattern		$s_{old} := s; S(\alpha_{discr}); \text{if}(cond(s, s_{old}))\{store(v_1, ...v_n)\}; \dots S(\alpha_{cont});$
Robot (delay)	$\mathtt{RL}_{\mathtt{r}t_{\Delta}\circ}$	$t_{\Delta,\mathtt{old}} := t_{\Delta}; t_{\Delta} := *; ?t_{\Delta} \leq t_{\Delta,\mathtt{high}}; \text{if}(c \geq T_S + t_{\Delta})\{c := 0; \vec{v}_{\mathtt{r}} := *; ?HC_r; \}$
		$\text{if}(t_{\Delta} > t_{\Delta,\mathtt{low}} \land t_{\Delta,\mathtt{old}} \leq t_{\Delta,\mathtt{low}})\{t_{\mathtt{hs}} := t; d_{\mathtt{hs}} := d(\vec{p}_{\mathtt{a}}, \vec{p}_{\mathtt{r}}); \}$
		$\text{if}(t_{\Delta} \leq t_{\Delta,\mathtt{low}} \land t_{\Delta,\mathtt{old}} > t_{\Delta,\mathtt{low}})\{t_{\mathtt{ls}} := t; d_{\mathtt{ls}} := d(\vec{p}_{\mathtt{a}}, \vec{p}_{\mathtt{r}}); \}$
Robot (noise)	$\mathtt{Sns}_{\mathtt{r}\eta\circ}$	$\eta_{\mathtt{old}} := \eta; \eta := *;\ ?\eta \leq \eta_{\mathtt{high}}; d_{sens} := d(\vec{p}_{\mathtt{r}}, \vec{p}_{\mathtt{a}}) + \eta;$
		$\text{if}(\eta > \eta_{\mathtt{low}} \land \eta_{\mathtt{old}} \leq \eta_{\mathtt{low}})\{t_{\mathtt{hs}} := t; d_{\mathtt{hs}} := d(\vec{p}_{\mathtt{a}}, \vec{p}_{\mathtt{r}}); \}$
Robot (drift)	$\mathtt{Plnt}_{\mathtt{r}\delta\circ}$	$\delta_{\mathtt{old}} := \delta; \delta := *; ?\delta \in [-\delta_{\mathtt{high}}, \delta_{\mathtt{high}}];$
		$\text{if}(\delta > \delta_{\mathtt{low}} \land \delta_{\mathtt{old}} \leq \delta_{\mathtt{low}})\{t_{\mathtt{hs}} := t; d_{\mathtt{hs}} := d(\vec{p}_{\mathtt{a}}, \vec{p}_{\mathtt{r}}); \}$
		$\{\dots \vec{p}_{\mathtt{r}}' = \vec{v}_{\mathtt{r}} \cdot \delta, \vec{p}_{\mathtt{a}}' = \vec{v}_{\mathtt{r}}\ \&\ c \leq T_S\}$

executed, $\mathtt{f}$ is set to 1 (otherwise $\mathtt{f} := 0;$). $\mathtt{f}_{\mathtt{old}}$ stores the previous failure state. If changes in the failure state occur ($cond(\mathtt{f}, \mathtt{f}_{\mathtt{old}})$) relevant system state variables, e.g., the time of failure and the current service level, are stored. $\mathtt{Flw}_{\mathtt{wf}\circ}$ in Table 7 demonstrates this observer pattern for the IWDS flow component with possible pump failures. $\mathtt{f}$ is set to 1 in the event of pump failures. Upon a failure ($\mathtt{f} > \mathtt{f}_{\mathtt{old}}$), we record the time ($t_{\mathtt{fail}}$), water level ($h_{\mathtt{fail}}$), and current supply ($sup_{rl\,\mathtt{fail}}$). We also observe recovery by checking if $\mathtt{f} < \mathtt{f}_{\mathtt{old}}$, and store the repair time ($t_{\mathtt{repair}}$).

The observer pattern for discrete noise, delay, and continuous noise shown in the lower part of Table 7 is similar to the observer pattern for failures. For these stressors we observe the variable s that introduces stress (η in case of noise and t_{Δ} in case of delay). We introduce an observer variable s_{old} to store the previous stress level. Then, we execute the discrete part of the HP of the stressed component ($S(\alpha_{discr})$). We check for possible disruptions via checking $cond(s, s_{old})$, e.g., whether one of the thresholds defined for high stress or low stress was just exceeded. In this case, we store relevant state variables and then execute continuous evolutions under stress ($S(\alpha_{cont})$).

$\mathtt{RL}_{\mathtt{r}t_{\Delta}\circ}$ in Table 7 shows the application to the RL agent of the robot in the presence of delay. First, the previous delay t_{Δ} is stored in $t_{\Delta,\mathtt{old}}$. Then, the RL agent is executed with a newly chosen delay. We check whether the delay level has switched from low to high stress ($t_{\Delta} > t_{\Delta,\mathtt{low}} \land t_{\Delta,\mathtt{old}} \leq t_{\Delta,\mathtt{low}}$). If so, we store the current time in $t_{\mathtt{hs}}$ and the distance in $d_{\mathtt{hs}}$. We also check whether the delay has switched back from high to low and store the time in $t_{\mathtt{ls}}$ and the distance in $d_{\mathtt{ls}}$. $\mathtt{Sns}_{\mathtt{r}\eta\circ}$ and $\mathtt{Plnt}_{\mathtt{r}\delta\circ}$ in Table 7 show the application for sensor noise and drift respectively. There we observe switching from low to high stress only.

Table 8. System Specification using Observer Patterns

$pre \longrightarrow [\{\mathtt{RL}_{\mathtt{w}};\mathtt{Flw}_{\mathtt{w}_{\mathtt{f}\circ}};\mathtt{Plnt}_{\mathtt{w}}\}^{*}]$ (IWDS fail)	Steps
(a) $i = 0 \wedge t - t_{\mathtt{fail}} \leq t_{\mathtt{max,fail}} \wedge sup_{rl}{}_{\mathtt{fail}} \geq sup_{\mathtt{max}} \wedge h_{\mathtt{fail}} > h_{\mathtt{max}} \longrightarrow sup_{rl} \geq sup_{\mathtt{dgr}} \wedge h \geq h_{min}$	95
(b) $i = i_{\mathtt{max}} \wedge t - t_{\mathtt{repair}} \geq t_{\mathtt{repair,dgr}} \longrightarrow sup_{rl} \geq sup_{\mathtt{dgr}} \wedge h \geq h_{\mathtt{dgr}}$	169
(c) $i = i_{\mathtt{max}} \wedge t - t_{\mathtt{repair}} \geq t_{\mathtt{repair,max}} \longrightarrow sup_{rl} \geq sup_{\mathtt{max}} \wedge h \geq h_{\mathtt{max}}$	259
(d) $sup_{rl} \geq 0 \wedge h \geq h_{min}$	90
$pre \longrightarrow [\{\mathtt{Sns}_{\mathtt{r}};\mathtt{RL}_{\mathtt{r}_{\mathtt{t}_{\Delta}\circ}};\mathtt{Opp}_{\mathtt{r}};\mathtt{Plnt}_{\mathtt{r}}\}^{*}]$ (Robot delay)	Steps
(e) $t_{\mathtt{hs}} = -1 \longrightarrow (\vec{v}_{\mathtt{r}} \geq v_{\mathtt{min,r}} \wedge d(\vec{p}_{\mathtt{a}},\vec{p}_{\mathtt{r}}) > \theta_{\mathtt{evd}} \vee \vec{v}_{\mathtt{r}} = 0)$	86
(f) $t_{\mathtt{hs}} \geq 0 \longrightarrow (\vec{v}_{\mathtt{r}} \geq v_{\mathtt{min,r}} \wedge d(\vec{p}_{\mathtt{a}},\vec{p}_{\mathtt{r}}) > \theta_{\mathtt{stp}} \vee \vec{v}_{\mathtt{r}} = 0)$	46
(g) $t_{\mathtt{hs}} \geq 0 \wedge t_{\Delta} \leq t_{\Delta,\mathtt{low}} \wedge d_{\mathtt{ls}} > \theta_{\mathtt{evd}} \longrightarrow (\vec{v}_{\mathtt{r}} \geq v_{\mathtt{min,r}} \wedge d(\vec{p}_{\mathtt{a}},\vec{p}_{\mathtt{r}}) > \theta_{\mathtt{evd}} \vee \vec{v}_{\mathtt{r}} = 0)$	77
$pre \longrightarrow [\{\mathtt{Sns}_{\mathtt{r}_{\eta\circ}};\mathtt{RL}_{\mathtt{r}};\mathtt{Opp}_{\mathtt{r}};\mathtt{Plnt}_{\mathtt{r}}\}^{*}]$ (Robot noise)	Steps
(h) $t_{\mathtt{hs}} = -1 \longrightarrow (\vec{v}_{\mathtt{r}} \geq v_{\mathtt{min,r}} \wedge d(\vec{p}_{\mathtt{a}},\vec{p}_{\mathtt{r}}) > \theta_{\mathtt{evd}} \vee \vec{v}_{\mathtt{r}} = 0)$	70
(i) $t_{\mathtt{hs}} \geq 0 \longrightarrow (\vec{v}_{\mathtt{r}} \geq v_{\mathtt{min,r}} \wedge d(\vec{p}_{\mathtt{a}},\vec{p}_{\mathtt{r}}) > \theta_{\mathtt{stp}} \vee \vec{v}_{\mathtt{r}} = 0)$	62
$pre \longrightarrow [\{\mathtt{Sns}_{\mathtt{r}};\mathtt{RL}_{\mathtt{r}};\mathtt{Opp}_{\mathtt{r}};\mathtt{Plnt}_{\mathtt{r}_{\delta\circ}}\}^{*}]$ (Robot drift)	Steps
(j) $t_{\mathtt{hs}} \geq 0 \longrightarrow (\vec{v}_{\mathtt{r}} \geq v_{\mathtt{min,r}} \wedge d(\vec{p}_{\mathtt{a}},\vec{p}_{\mathtt{r}}) > \theta_{\mathtt{stp}} \vee \vec{v}_{\mathtt{r}} = 0)$	90

4 Evaluation

To evaluate our approach, we have defined resilience properties for our two case studies, which ensure that the system reacts to different stress conditions with an appropriate service level. Table 8 shows the specifications for both the IWDS and the robot. The RL components use the embeddings defined in Table 6.

For the IWDS, we have specified the following resilience properties in the presence of pump failures (IWDS fail):

(a) If the pump has failed for less than $t_{\mathtt{max,fail}}$ time and the system was at full service level when the pump failed, degraded service is still provided.
(b) If the pump is functioning for $t_{\mathtt{repair,dgr}}$, at least degraded service is provided.
(c) If the pump is functioning for $t_{\mathtt{repair,max}}$ time, full service is provided.
(d) A minimum water level h_{min} is always maintained, even with no supply.

For the autonomous robot, we have specified the following resilience properties in the presence of delay (e-g), discrete noise (h-i), and drift (j).

(e) If high delay has never occurred ($t_{\mathtt{hs}} = -1$) the robot either maintains $\theta_{\mathtt{evd}}$ while moving or stops (if the opponent gets too close).
(f) If high delay occurred at some point ($t_{\mathtt{hs}} \geq 0$) the robot stops before $\theta_{\mathtt{stp}}$.
(g) If high delay occurred ($t_{\mathtt{hs}} \geq 0$) and the delay returned to low delay ($t_{\Delta} \leq t_{\Delta,\mathtt{low}}$) at a distance greater than the evasion distance ($d_{\mathtt{hs}} > \theta_{\mathtt{evd}}$), the robot either maintains $\theta_{\mathtt{evd}}$ while moving or stops (if the opponent gets too close).
(h) If high sensor noise has never occurred ($t_{\mathtt{hs}} = -1$), the robot either maintains $\theta_{\mathtt{evd}}$ while moving or stops (if the opponent gets too close).

(i) If high sensor noise was experienced ($t_{\text{hs}} \geq 0$), the robot stops before $\theta_{\texttt{stp}}$.
(j) If severe drift was experienced ($t_{\text{hs}} \geq 0$), the robot stops before reaching $\theta_{\texttt{stp}}$.

We have verified all of the resilience properties defined above for the IWDS and the autonomous robot using KeYmaera X. The models and proof files are available at https://www.uni-muenster.de/EmbSys/research/Simulink2dL.html. The last column of Table 8 shows the number of manual proof steps used for each system specification, providing a rough comparison of verification effort. More proof steps generally indicate higher effort, but shorter proofs may also result from better rule application or clever invariant choices. The proofs for properties (b) with 169 steps and (c) with 259 steps were the most challenging and took roughly two and three person-days respectively. All other proofs were completed in less than one person-day each.

5 Related Work

Many approaches for formal verification of hybrid systems are based on model checking and leverage symbolic reachability, e.g., with polyhedra [13], zonotopes [25], or support functions [35] to over-approximate continuous state spaces. However, these approaches are usually limited to linear dynamics or only consider time-bounded properties. Probabilistic methods [20,26,27,39] face similar limitations. Approaches for deductive verification of hybrid systems, such as differential dynamic logic [55,56], differential Hoare logic [21], and hybrid Hoare logic for HCSP [38] exploit inductive reasoning to overcome the state space explosion problem. However, they typically require high manual effort and expertise.

Many approaches for formal verification of systems modeled in Simulink, e.g., [7,30,58], including the Simulink Design Verifier [46], only support discrete models. Methods that support models of hybrid systems are, e.g., proposed in [12,14,48,66]. However, none of these methods supports learning or resilience.

There also exists a broad variety of approaches to formally ensure safety of learning components using shielding or runtime monitoring [5,19,33,54]. KeYmaera X also enables synthesis of verified runtime monitors for learning components [23,49–51]. However, these methods do not cover the integration of stressors through reusable patterns or dynamic adaptations via service levels.

A wide range of research exists on proof reusability, e.g., within KeY [9] and KeYmaera X [22]. There exist various approaches for automated invariant generation [31,59,63]. However, these approaches focus on proof construction rather than domain-specific patterns [28]. [24,61] provide application-specific patterns to address complex verification issues, like structured arrays [24] and parallel prefix sums [61]. A contract-based approach for system analysis across application domains is introduced in [15], and techniques for structured proof reuse in software variations are proposed in [64], while [6,32,65] focus on easing verification through reusable patterns. [53] introduces a contract-based verification in d$\mathcal{L}$ and provides specifications for output and communication reliability.

There exists work on definitions and formal verification of resilience for CPS, e.g., using temporal logics [11,60,62] but these often only consider time discrete systems. [29,52] consider resilience and robustness for timed (I/O) systems. [10, 18,40] use Markov decision processes or discrete time Markov chains. However, none of these approaches support deductive verification of hybrid systems.

To the best of our knowledge, none of the existing works specifically address resilience in AHS. In our own previous work [1–3], we have presented some initial concepts for reusable resilience contracts. However, we only provided a rudimentary service level concept and we have not considered reusable resilience specifications using stressor and observer patterns.

6 Conclusion and Outlook

In this paper, we have presented an approach for the formal specification and verification of resilience in AHS using d$\mathcal{L}$ and the interactive theorem prover KeYmaera X. We have presented a structured approach and reusable patterns for modeling stressors and observers, and for specifying resilience as a service level response. Our specifications are more dynamic than traditional safety properties, and thus better capture the adaptive aspects of resilience, as we define service levels relative to the intensity of stress an autonomous hybrid system experiences. By providing resilience contract patterns for learning components, we enable safe and resilient learning with a shielding-based approach, where the shields can automatically be generated from our resilience contracts.

We have demonstrated the applicability of our approach with designs inspired by MathWorks, namely an intelligent water distribution system and an autonomous robot. Our patterns can help reduce the specification effort for deductive verification of resilience properties. By defining coarse- or fine-granular service levels, the designer can choose a trade-off between the specification and verification effort and the strengths of the resulting resilience guarantees.

In future work, we plan to fully integrate our reusable patterns and specifications for resilience with our existing Simulink2d$\mathcal{L}$ transformation to enable the automatic transformation of Simulink models into resilient AHS models in d$\mathcal{L}$. We plan to use our approach on larger case studies and to investigate the trade-off between fine- and coarse-granular service levels, and their effect on the precision and permissiveness of contracts. We plan to derive guidelines for the specification of service levels together with safety thresholds for various system classes. Furthermore, we plan to exploit symbolic AI and explainability techniques to generate such specifications automatically.

References

1. Adelt, J., Brettschneider, D., Herber, P.: Reusable contracts for safe integration of reinforcement learning in hybrid systems. In: Bouajjani, A., Holík, L., Wu, Z. (eds.) Automated Technology for Verification and Analysis. ATVA 2022. LNCS, vol. 13505. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19992-9_4
2. Adelt, J., Bruch, S., Herber, P., Niehage, M., Remke, A.: Shielded Learning for Resilience and Performance Based on Statistical Model Checking in Simulink. In: Steffen, B. (eds.) Bridging the Gap Between AI and Reality, vol. 14380, pp. 94–118. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-46002-9_6
3. Adelt, J., Herber, P., Niehage, M., Remke, A.: Towards safe and resilient hybrid systems in the presence of learning and uncertainty. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles. ISoLA 2022. LNCS, vol. 13701, pp. 299–319. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19849-6_18
4. Adelt, J., Liebrenz, T., Herber, P.: Formal verification of intelligent hybrid systems that are modeled with Simulink and the reinforcement learning toolbox. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 349–366. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_19
5. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: AAAI Conference on Artificial Intelligence **32** (2018). https://doi.org/10.1609/aaai.v32i1.11797
6. André, É.: Observer patterns for real-time systems. In: 2013 18th International Conference on Engineering of Complex Computer Systems, pp. 125–134. IEEE Computer Society (2013). https://doi.org/10.1109/ICECCS.2013.26
7. Araiza-Illan, D., Eder, K., Richards, A.: Formal verification of control systems properties with theorem proving. In: International Conference on Control (CONTROL), pp. 244–249. IEEE (2014). https://doi.org/10.1109/CONTROL.2014.6915147
8. Arghandeh, R., Von Meier, A., Mehrmanesh, L., Mili, L.: On the definition of cyber-physical resilience in power systems. Renew. Sustain. Energy Rev. **58**, 1060–1069 (2016). https://doi.org/10.1016/j.rser.2015.12.193
9. Beckert, B., Klebanov, V.: Proof reuse for deductive program verification. In: International Conference on Software Engineering and Formal Methods (SEFM), pp. 77–86. IEEE (2004). https://doi.org/10.1109/SEFM.2004.1347505
10. Camilli, M., Mirandola, R., Scandurra, P.: Runtime equilibrium verification for resilient cyber-physical systems. In: 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), pp. 71–80. IEEE (2021). https://doi.org/10.1109/ACSOS52086.2021.00025
11. Chen, H., Lin, S., Smolka, S.A., Paoletti, N.: An STL-based formulation of resilience in cyber-physical systems. In: Formal Modeling and Analysis of Timed Systems: 20th International Conference, FORMATS 2022, Warsaw, Poland, 13–15 September 2022, Proceedings, pp. 117–135. Springer-Verlag, Heidelberg (2022). https://doi.org/10.1007/978-3-031-15839-1_7
12. Chen, M., et al.: MARS: a toolchain for modelling, analysis and verification of hybrid systems. In: Hinchey, M.G., Bowen, J.P., Olderog, E.-R. (eds.) Provably Correct Systems. NMSSE, pp. 39–58. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-48628-4_3

13. Chutinan, A., Krogh, B.H.: Computing polyhedral approximations to flow pipes for dynamic systems. In: Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No. 98CH36171), vol. 2, pp. 2089–2094. IEEE (1998). https://doi.org/10.1109/CDC.1998.758642
14. Chutinan, A., Krogh, B.H.: Computational techniques for hybrid system verification. IEEE Trans. Autom. Control **48**(1), 64–75 (2003). https://doi.org/10.1109/TAC.2002.806655
15. Cimatti, A., Dorigatti, M., Tonetta, S.: OCRA: a tool for checking the refinement of temporal contracts. In: International Conference on Automated Software Engineering, pp. 702–705. IEEE (2013). https://doi.org/10.1109/ASE.2013.6693137
16. Clark, A., Zonouz, S.: Cyber-physical resilience: definition and assessment metric. IEEE Trans. Smart Grid **10**(2), 1671–1684 (2017). https://doi.org/10.1109/TSG.2017.2776279
17. Cloth, L., Haverkort, B.R.: Model checking for survivability! In: International Conference on the Quantitative Evaluation of Systems (QEST), pp. 145–154. IEEE (2005). https://doi.org/10.1109/QEST.2005.21
18. Cámara, J., de Lemos, R.: Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In: 2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 53–62 (2012). https://doi.org/10.1109/SEAMS.2012.6224391
19. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal Stratego. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 206–211. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_16
20. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Wang, Z.: Time for statistical model checking of real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 349–355. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_27
21. Foster, S., Huerta y Munive, J.J., Struth, G.: Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In: Fahrenberg, U., Jipsen, P., Winter, M. (eds.) RAMiCS 2020. LNCS, vol. 12062, pp. 169–186. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43520-2_11
22. Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 527–538. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_36
23. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: toward safe control through proof and learning. AAAI Conf. on Artif. Intellig. **32** (2018). https://doi.org/10.1609/aaai.v32i1.12107
24. Genestier, R., Giorgetti, A., Petiot, G.: Sequential generation of structured arrays and its deductive verification. In: Blanchette, J.C., Kosmatov, N. (eds.) TAP 2015. LNCS, vol. 9154, pp. 109–128. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21215-9_7
25. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 291–305. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31954-2_19
26. Gribaudo, M., Remke, A.: Hybrid petri nets with general one-shot transitions. Perform. Eval. **105**, 22–50 (2016). https://doi.org/10.1016/J.PEVA.2016.09.002
27. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods Syst. Des. **43**(2), 191–232 (2013). https://doi.org/10.1007/S10703-012-0167-Z

28. Hähnle, R., Huisman, M.: Deductive software verification: from pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 345–373. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_18
29. Henzinger, T.A., Otop, J., Samanta, R.: Lipschitz robustness of timed I/O systems. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 250–267. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_12
30. Herber, P., Reicherdt, R., Bittner, P.: Bit-precise formal verification of discrete-time MATLAB/Simulink models using SMT solving. In 2013 Proceedings of the International Conference on Embedded Software, pp. 1–10. IEEE (2013). https://doi.org/10.1109/EMSOFT.2013.6658586
31. Hoder, K., Kovács, L., Voronkov, A.: Invariant generation in vampire. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 60–64. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_7
32. Jonker, C.M., Treur, J., de Vries, W.: Reuse and abstraction in verification: agents acting in dynamic environments. In: Ciancarini, P., Wooldridge, M.J. (eds.) AOSE 2000. LNCS, vol. 1957, pp. 253–267. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44564-1_17
33. Könighofer, B., Lorber, F., Jansen, N., Bloem, R.: Shield synthesis for reinforcement learning. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12476, pp. 290–306. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_16
34. Laprie, J.C.: From dependability to resilience. In: 38th IEEE/IFIP International Conference On dependable systems and networks, pp. G8–G9 (2008)
35. Le Guernic, C., Girard, A.: Reachability analysis of hybrid systems using support functions. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 540–554. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_40
36. Liebrenz, T., Herber, P., Glesner, S.: Deductive verification of hybrid control systems modeled in simulink with KeYmaera X. In: Sun, J., Sun, M. (eds.) ICFEM 2018. LNCS, vol. 11232, pp. 89–105. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02450-5_6
37. Liebrenz, T., Herber, P., Glesner, S.: A service-oriented approach for decomposing and verifying hybrid system models. In: Arbab, F., Jongmans, S.-S. (eds.) FACS 2019. LNCS, vol. 12018, pp. 127–146. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-40914-2_7
38. Liu, J., et al.: A calculus for hybrid CSP. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 1–15. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_1
39. Lygeros, J., Prandini, M.: Stochastic hybrid systems: a powerful framework for complex, large scale applications. Eur. J. Control. **16**(6), 583–594 (2010). https://doi.org/10.3166/EJC.16.583-594
40. Madni, A.M., Erwin, D., Sievers, M.: Constructing models for systems resilience: challenges, concepts, and formal methods. MDPI Syst. **8**(1), 3 (2020). https://doi.org/10.3390/systems8010003
41. MathWorks: control and simulate multiple warehouse robots. https://www.mathworks.com/help/robotics/ug/control-and-simulate-multiple-warehouse-robots.html
42. MathWorks: MATLAB simulink. www.mathworks.com/products/simulink.html
43. MathWorks: reinforcement learning toolbox. https://www.mathworks.com/products/reinforcement-learning.html

44. MathWorks: robotics systems toolbox. https://www.mathworks.com/products/robotics.html
45. MathWorks: simulink. https://www.mathworks.com/products/simulink.html
46. MathWorks: simulink design verifier. https://www.mathworks.com/products/simulink-design-verifier.html
47. MathWorks: Water distribution system scheduling using reinforcement learning. https://www.mathworks.com/help/reinforcement-learning/ug/water-tank-simulink-reinforcement-learning-environment.html
48. Minopoli, S., Frehse, G.: SL2SX Translator: from Simulink to SpaceEx models. In: International Conference on Hybrid Systems: Computation and Control, pp. 93–98. ACM (2016). https://doi.org/10.1145/2883817.2883826
49. Mitsch, S., Platzer, A.: ModelPlex: verified runtime validation of verified cyber-physical system models. Formal Methods Syst. Des. **49**(1), 33–74 (2016). https://doi.org/10.1007/s10703-016-0241-z
50. Mitsch, S., Platzer, A.: Verified runtime validation for partially observable hybrid systems. CoRR (2018). https://doi.org/10.48550/arXiv.1811.06502
51. Mitsch, S., Platzer, A.: The KeYmaera X Proof IDE - concepts on usability in hybrid systems theorem proving. Electron. Proc. Theoret. Comput. Sci. **240** (2017). https://doi.org/10.4204/EPTCS.240.5
52. Mouelhi, S., Laarouchi, M.E., Cancila, D., Chaouchi, H.: Predictive formal analysis of resilience in cyber-physical systems. IEEE Access **7**, 33741–33758 (2019). https://doi.org/10.1109/ACCESS.2019.2903153
53. Müller, A., Mitsch, S., Retschitzegger, W., Schwinger, W., Platzer, A.: Tactical contract composition for hybrid system component verification. Int. J. Softw. Tools Technol. Transfer **20**(6), 615–643 (2018). https://doi.org/10.1007/s10009-018-0502-9
54. Phan, D., et al.: A component-based simplex architecture for high-assurance cyber-physical systems. In: International Conference on Application of Concurrency to System Design, pp. 49–58. IEEE (2017). https://doi.org/10.1109/ACSD.2017.23
55. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reason. **41**(2), 143–189 (2008). https://doi.org/10.1007/s10817-008-9103-8
56. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. J. Autom. Reason. **59**(2), 219–265 (2017). https://doi.org/10.1007/s10817-016-9385-1
57. Platzer, A.: The complete proof theory of hybrid systems. In: 2012 27th Annual IEEE Symposium on Logic in Computer Science, pp. 541–550 (2012). https://doi.org/10.1109/LICS.2012.64
58. Reicherdt, R., Glesner, S.: Formal verification of discrete-time MATLAB/Simulink models using boogie. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 190–204. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_14
59. Rodríguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Sci. of Comp. Progr. **64**(1), 54–75 (2007). https://doi.org/10.1016/j.scico.2006.03.003
60. Rungger, M., Tabuada, P.: A notion of robustness for cyber-physical systems. IEEE Trans. Autom. Control **61**(8), 2108–2123 (2015). https://doi.org/10.1109/TAC.2015.2492438
61. Safari, M., Oortwijn, W., Joosten, S., Huisman, M.: Formal verification of parallel prefix sum. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NFM 2020. LNCS, vol. 12229, pp. 170–186. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_10

62. Saoud, A., Jagtap, P., Soudjani, S.: Temporal logic resilience for cyber-physical systems. In: 2023 62nd IEEE Conference on Decision and Control (CDC), pp. 2066–2071 (2023). https://doi.org/10.1109/CDC49753.2023.10384033, https://api.semanticscholar.org/CorpusID:267046171
63. Sogokon, A., Mitsch, S., Tan, Y.K., Cordwell, K., Platzer, A.: Pegasus: a framework for sound continuous invariant generation. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 138–157. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_10
64. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: Family-based deductive verification of software product lines. In: International Conference on Generative Programming and Component Engineering, pp. 11–20. ACM (2012). https://doi.org/10.1145/2371401.2371404
65. Vogel, T., Carwehl, M., Rodrigues, G.N., Grunske, L.: A property specification pattern catalog for real-time system verification with UPPAAL. Inf. Softw. Technol. **154**, 107100 (2023). https://doi.org/10.1016/j.infsof.2022.107100
66. Zou, L., Zhan, N., Wang, S., Fränzle, M.: Formal verification of Simulink/Stateflow diagrams. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 464–481. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24953-7_33

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Switching Controller Synthesis for Hybrid Systems Against STL Formulas

Han Su[1,2], Shenghua Feng[3(✉)], Sinong Zhan[4], and Naijun Zhan[1,5]

[1] State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
{suhan,znj}@ios.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
[3] Zhongguancun Laboratory, Beijing, China
fengsh@zgclab.edu.cn
[4] Department of Electrical and Computer Engineering, Northwestern University, Evanston, USA
SinongZhan2028@u.northwestern.edu
[5] School of Computer Science, Peking University, Beijing, China
znj@ios.ac.cn

Abstract. Switching controllers play a pivotal role in directing hybrid systems (HSs) towards the desired objective, embodying a "correct-by-construction" approach to HS design. Identifying these objectives is thus crucial for the synthesis of effective switching controllers. While most of existing works focus on safety and liveness, few of them consider timing constraints. In this paper, we delves into the synthesis of switching controllers for HSs that meet system objectives given by a fragment of STL, which essentially corresponds to a reach-avoid problem with timing constraints. Our approach involves iteratively computing the state sets that can be driven to satisfy the reach-avoid specification with timing constraints. This technique supports to create switching controllers for both constant and non-constant HSs. We validate our method's soundness, and confirm its relative completeness for a certain subclass of HSs. Experiment results affirms the efficacy of our approach.

Keywords: Hybrid Systems · Switching Controller Synthesis · Signal Temporal Logic · Reach-Avoid

1 Introduction

Hybrid systems (HSs) provide a robust mathematical specification in modeling cyber-physical systems (CPS) with their unique fusion of continuous physical

This work has been partially funded by the NSFC under grant No. 62192732 and 62032024, by the National Key R&D Program of China under grant No. 2022YFA1005101, by the CAS Project for Young Scientists in Basic Research under grant No. YSBR-040.

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 229–247, 2025.
https://doi.org/10.1007/978-3-031-71177-0_15

dynamics and discrete switching behaviors. Many CPSs are often complex and safety-critical which necessitates intricate control specifications. Switching controller synthesis offers a formal guarantee of the given specification of HS. Its applications include attitude control in aerospace [3], aircraft collision-avoidance protocols in avionics [43], and pacemakers for treating bradycardia [53], etc.

With the escalating complexity of CPSs [44,45,54], the specifications required to ensure their proper functionality grow increasingly intricate. Among these, the importance of timing constraints becomes paramount [48,49]. This is evident in various scenarios, from orchestrating synchronized reactions in chemical processing [12] to ensuring seamless operations in multi-robot systems [24]. In this context, Signal Temporal Logic (STL), a rigorous formalism for defining linear-time properties of continuous signals [27], is exceptionally well-suited for specifying intricate timing constraints and qualitative properties of CPSs.

However, switching controller synthesis for HSs against STL specifications is not well addressed in the literature. The primary challenge arises from the complex interactions between continuous behaviors and discrete transitions. A common technique to synthesize switching controllers for HSs with complex specifications is the abstraction-based method [26,28]. This technique involves abstracting the continuous state space of each mode into a finite set of states, which often results in the loss of precise timing information for each mode. Consequently, the abstraction-based technique struggles with timing constraint analysis in the abstracted state space. In contrast, Mixed Integer Linear Programming (MILP) based technique [34] for switching controller synthesis against STL specification can provide precise timing information, but this method faces challenges in handling the intricate interactions of diverse discrete transitions between modes.

In this paper, we considered the switching controller synthesis problem for HSs against a fragment of STL specification, which essentially corresponds to a reach-avoid problem with timing constraints. To the best of our knowledge, this is the first work that uses STL to specify HSs with both discrete transitions and continuous dynamics. Similar work in [39] focused only on HSs with discrete time dynamics in each mode, significantly simplifying the problem. The key idea behind our approach involves iteratively computing a sequence of state-time sets (x, t), state x and time t. These sets ensure that an HS, starting from state x at time t, adheres to the STL specification within a certain number (i.e., the number of iterations) of switches. The state-time sets are computed explicitly when the dynamics of the HSs are constant, and are inner-approximated when the dynamics are non-constant. Based on the state-time sets, we propose a sound and relatively complete method to synthesize a switching controller that satisfies the STL specification. Our experimental results demonstrate the efficacy of this approach.

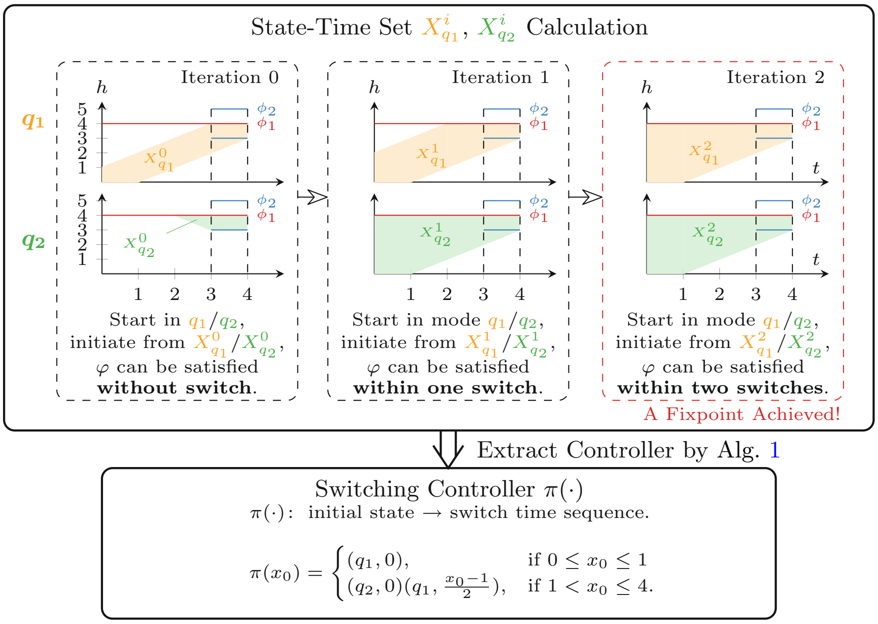

State-Time Set $X^i_{q_1}$, $X^i_{q_2}$ Calculation
Iteration 0
Iteration 1
Iteration 2
q_1
q_2
h
t
ϕ_1
ϕ_2
$X^0_{q_1}$
$X^0_{q_2}$
$X^1_{q_1}$
$X^1_{q_2}$
$X^2_{q_1}$
$X^2_{q_2}$
Start in q_1/q_2, initiate from $X^0_{q_1}/X^0_{q_2}$, φ can be satisfied **without switch**.
Start in mode q_1/q_2, initiate from $X^1_{q_1}/X^1_{q_2}$, φ can be satisfied **within one switch**.
Start in mode q_1/q_2, initiate from $X^2_{q_1}/X^2_{q_2}$, φ can be satisfied **within two switches**.
A Fixpoint Achieved!
Extract Controller by Alg. 1
Switching Controller $\pi(\cdot)$
$\pi(\cdot)$: initial state $\rightarrow$ switch time sequence.
$\pi(x_0) = \begin{cases} (q_1, 0), & \text{if } 0 \le x_0 \le 1 \\ (q_2, 0)(q_1, \frac{x_0 - 1}{2}), & \text{if } 1 < x_0 \le 4. \end{cases}$

Fig. 1. An overview of our method

The main contributions can be summarized as follows: (i) We conceptualize *state-time set* for HSs. (ii) We propose a methodology to synthesize switching controllers for HSs against a fragment of STL specification. (iii) We develop a prototype to demonstrate the efficiency and practical applicability of our methodology.

Organization. Section 2 gives an overview of our approach, Sect. 3 provides a recap of important preliminaries and formally defines the problem. We illustrate the calculation of the state-time sets in Sect. 4. Based on the state-time sets, Sect. 5 shows how to derive switching systems against a STL specification. In Sect. 6, we demonstrate the efficacy of our method through several examples. We discuss related work in Sect. 7 and draw conclusion in Sect. 8.

Due to space restrictions, proofs and benchmark details have been omitted, which can be found in an extended version of this paper [40]. Source code and examples of this paper can be found at Figshare: https://doi.org/10.6084/m9.figshare.26057410.v1.

2 An Illustrative Prelude

Example 1. In the reactor system depicted in Fig. 2, liquid is continuously consumed by the reaction and is replenished through pipe P. The system alternates between modes of adding liquid ($\boldsymbol{q_1}$) and exclusively consuming it ($\boldsymbol{q_2}$). The objectives are to keep the liquid level, h, between 0 and 4 meters, and to ensure that h remains between 3 and 5 meters at a certain point during a critical reaction phase - time interval 3 to 4, for proper interaction with the reactor rod R. These objectives can be given as an STL formula $\varphi = (0 \le h \le 4)\,\mathcal{U}_{[3,4]}(3 \le h \le 5)$. $\triangleleft$

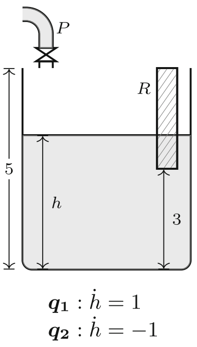

P
R
5
h
3
$\boldsymbol{q_1} : \dot{h} = 1$
$\boldsymbol{q_2} : \dot{h} = -1$

Fig. 2. Reactor System

We present the core idea behind our approach in Fig. 1. Initially, we compute the state-time set X_q^i iteratively. As shown in the upper block of Fig. 1, the set X_q^i encompasses states in mode q from which φ can be satisfied within i switches (as detailed in Sect. 5). Once these *state-time sets* are determined, the switching controllers can be synthesized using the methods outlined in Alg. 1 and Alg. 2.

3 Notations and Problem Formulation

Notations. Let $\mathbb{N}, \mathbb{R}$, and $\mathbb{R}_{\ge 0}$ denote the set of natural, real, and non-negative real numbers, respectively. Given vector $x \in \mathbb{R}^n$, x_i refers to its i-th component, and $p\,[x = u]$ denotes the replacement of x by u for any predicate p where x serves as a variable.

Differential Dynamics. We consider a class of dynamical systems featuring differential dynamics governed by ordinary differential equations (ODEs) of the form $\dot{\boldsymbol{x}} = f(\boldsymbol{x})$, where f is a continuous differentiable function. Given an initial state $x_0 \in \mathbb{R}^n$, there exist a unique solution $\boldsymbol{x} : \mathbb{R}_{\ge 0} \to \mathbb{R}^n$ in the sense that $\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t))$ for $t \ge 0$ and $\boldsymbol{x}(0) = x_0$.

Switched Systems. A switched system is defined as a tuple $\Phi = (Q, F, \texttt{Init}, \pi)$, where

- $Q \triangleq \{\, q_1, q_2, \ldots, q_m \,\}$ is a finite set of discrete modes.
- $F \triangleq \{\, f_q \mid q \in Q \,\}$ is a set of vector fields, and each mode $q \in Q$ endows with a unique vector field f_q which specifies how system evolves in mode q.
- $\texttt{Init} \subseteq \mathbb{R}^n$ is a set of initial states.
- $\pi \colon \texttt{Init} \to (\mathbb{R}_{\ge 0} \to Q)$ is a switching controller. The controller maps each initial state $x_0 \in \texttt{Init}$ to a piecewise constant function $\pi(x_0)$, which in turn maps a time t to the corresponding control mode $\pi(x_0)(t)$.

Given any initial state x_0, the dynamics of the switched system Φ is governed by equation $\dot{\boldsymbol{x}}(t) = f_{\pi(x_0)(t)}(\boldsymbol{x}(t))$ with initial condition $\boldsymbol{x}(0) = x_0$.

Signal Temporal Reach-Avoid. We consider a fragment of signal temporal logic, namely *signal temporal reach-avoid* formula (ST-RA for short). The syntax of ST-RA is defined by

$$\begin{aligned}\phi ::= &\ \mu(x,t) \geq 0 \mid \neg\phi \mid \phi \wedge \phi \\ \varphi ::= &\ \phi_1 \, \mathcal{U}_I \, \phi_2\end{aligned}$$

where ϕ is a Boolean combination of predicates over x and time t, $I \triangleq [l,u]$ is a closed time interval for some $0 \leq l \leq u$. Intuitively, an ST-RA formula φ expresses the requirement that the system should reach ϕ_2 while avoid leaving ϕ_1 within time frame I. The semantics of ST-RA formula, in alignment with STL, is defined as the satisfaction of a formula φ with respect to a signal $\boldsymbol{x}$ and a time instant t.

Remark 1. Compared with the standard signal temporal logic (STL) [27], ST-RA formula does not allow nested "until" operator, this makes ST-RA formula a fragment of STL.

Formally, given function $\boldsymbol{x}\colon \mathbb{R}_{\geq 0} \to \mathbb{R}^n$ (termed signal) and time τ, the satisfaction of φ at $(\boldsymbol{x},\tau)$, denoted by $(\boldsymbol{x},\tau) \models \varphi$, is inductively defined as follows:

$$\begin{aligned}
&(\boldsymbol{x},\tau) \models \mu(x,t) \geq 0 && \text{iff} \quad \mu(\boldsymbol{x}(\tau),\tau) \geq 0\,; \\
&(\boldsymbol{x},\tau) \models \neg\phi && \text{iff} \quad (\boldsymbol{x},\tau) \not\models \phi\,; \\
&(\boldsymbol{x},\tau) \models \phi_1 \wedge \phi_2 && \text{iff} \quad (\boldsymbol{x},\tau) \models \phi_1 \text{ and } (\boldsymbol{x},\tau) \models \phi_2\,; \\
&(\boldsymbol{x},\tau) \models \phi_1 \, \mathcal{U}_I \, \phi_2 && \text{iff} \quad \exists \tau' \geq \tau, \text{ such that } \tau' - \tau \in I,\ (\boldsymbol{x},\tau') \models \phi_2\,, \\
& && \qquad\qquad \text{and } \forall \tau'' \in [\tau,\tau'], (\boldsymbol{x},\tau'') \models \phi_1\,.
\end{aligned}$$

Intuitively, the subscript I in the until operator $\mathcal{U}_I$ defines the timing constraints under which a signal must reach ϕ_2 while avoid leaving ϕ_1.

Given a ST-RA formula ϕ, we say a switched system $\Phi = (Q, F, \texttt{Init}, \pi)$ models φ, denoted by $\Phi \models \varphi$, if $(\boldsymbol{x}, 0) \models \varphi$ for any trajectory $\boldsymbol{x}$ starting from initial set Init. We now formulate the problem addressed in this paper.

Problem Formulation. Suppose there exists a finite set of control modes $Q = \{ q_1, q_2, \ldots, q_m \}$ and associated vector fields $F = \{ f_{q_1}, f_{q_2}, \ldots, f_{q_m} \}$. Each mode $q \in Q$ is associated with a vector field f_q that governs the system's behavior in mode q. Let $\varphi = \phi_1 \, \mathcal{U}_I \, \phi_2$ be an ST-RA formula, a natural question is how to design a system that incorporates mode $q \in Q$ as subsystems, while ensuring any trajectory of the system satisfies φ. To address this, we formulate the problem as follows.

Synthesis of Switched System. Given a finite set of discrete modes $Q = \{ q_1, q_2, \ldots, q_m \}$, a set of vector fields $F = \{ f_{q_1}, f_{q_2}, \ldots, f_{q_m} \}$, and a ST-RA formula $\varphi = \phi_1 \, \mathcal{U}_I \, \phi_2$, the switched system synthesis problem aims to synthesize a switched system $\Phi = (Q, F, \texttt{Init}, \pi)$, such that $\Phi \models \varphi$.

Remark 2. The solutions to the above synthesis problem is inherently non-unique and may encompass trivialities, such as the one only with an empty initial set. Therefore, our goal is to identify a system with a nontrivial initial set `Init`.

4 State-Time Set and Its Calculation

This section dedicates to synthesize a switched system Φ that satisfies the given ST-RA formula $\varphi = \phi_1 \mathcal{U}_I \phi_2$. The key idea behind our approach is to compute a sequence of state-time sets $\{X_q^i\}_{q\in Q}$ for $i \in \mathbb{N}$, where X_q^i denotes the set of all (x, τ) such that starting from x at time τ in mode q, the system can be driven to reach ϕ_2 while satisfying ϕ_1 *within i times of switches*. In what follows, we first formally propose the concept of state-time sets and show how to calculate it explicitly. Subsequently, leveraging these state-time sets, we demonstrate the synthesis of a switched system that satisfies φ in Sect. 5.

4.1 State-Time Sets

The concept of state-time set is formally summarised by the following definition.

Definition 1 (State-time Sets). *For any $i \in \mathbb{N}$ and any $q \in Q$, let X_q^i denote the set of all state-time pairs (x, τ) such that there exists a controller $\pi(x)\colon [\tau, \infty) \to Q$, satisfying*

(i) $\pi(x)(\tau) = q$, and the piecewise constant function $\pi(x)$ contains at most i discontinuous points;
(ii) $(\boldsymbol{x}, \tau) \models \phi_1 \mathcal{U}_{I \dot{-} \tau} \phi_2$, where $\boldsymbol{x}$ is the solution of ODE $\dot{\boldsymbol{x}}(t) = f_{\pi(x)(t)}(\boldsymbol{x}(t), t)$ over $[\tau, \infty)$ with $\boldsymbol{x}(\tau) = x$, and $I \dot{-} \tau \triangleq [l - \tau, u - \tau] \cap \mathbb{R}_{\geq 0}$ for any interval $I = [l, u]$.

Intuitively, condition (ii) suggests that the system can be driven to reach ϕ_2 while satisfying ϕ_1 from x at time τ, and condition (i) indicates that the system initially remains in mode q, and the switching controller undergoes no more than i switches. From the above definition of state-time sets, the following results can be derived:

Corollary 1. *The following properties hold for the state-time sets $\{X_q^i\}_{q\in Q}$:*

1. *For any $q \in Q$, $\{X_q^i\}$ is monotonically increasing, i.e. $X_q^0 \subseteq X_q^1 \subseteq X_q^2 \subseteq \cdots$.*
2. *For any $i \in \mathbb{N}$ and any $x \in X_q^i[t=0]$, x can be driven to satisfy $\phi_1 \mathcal{U}_I \phi_2$, i.e. there exists a switching controller π, such that $(\boldsymbol{x}, 0) \models \phi_1 \mathcal{U}_I \phi_2$, where $\boldsymbol{x}$ is the trajectory starting from x at time 0 under controller π, and $X_i^q[t=0] \triangleq \{x \mid (x, 0) \in X_i^q\}$ is the projection of X_i^q into $t = 0$.*
3. *$\cup_{i\in\mathbb{N}} \cup_{q\in Q} X_q^i[t=0]$ is the set of all states that can be driven to satisfy $\phi_1 \mathcal{U}_I \phi_2$.*

According to Cor. 1, the state-time sets encompass the initial set of the switched system that we intend to synthesize. However, the state-time set and controller defined in Def. 1 are not given explicitly. To address this, we first elucidate the process of calculating the state-time sets.

The subsequent result establishes a relationship between the sets $\{X_q^i\}_{q\in Q}$ and $\{X_q^{i-1}\}_{q\in Q}$, forming the foundation for the inductive computation of state-time sets.

Theorem 1. *Follow the notations as before, we have*[1]

1. *Given any $q \in Q$, $(x,\tau) \in X_q^0$ if and only if*
$$(\boldsymbol{x},\tau) \models \phi_1 \,\mathcal{U}\, (\phi_2 \wedge (t \in I)) \tag{1}$$
where $\boldsymbol{x}$ is the solution of ODE $\dot{\boldsymbol{x}}(t) = f_q(\boldsymbol{x}(t),t)$ over $[\tau,\infty)$ with $\boldsymbol{x}(\tau) = x$.
2. *Given any $q \in Q$, for any $i \geq 1$, $(x,\tau) \in X_q^i$ if and only if*
$$\exists q' \neq q \in Q,\ (\boldsymbol{x},\tau) \models \phi_1 \,\mathcal{U}\, X_{q'}^{i-1} \tag{2}$$
where $\boldsymbol{x}$ is the solution of ODE $\dot{\boldsymbol{x}}(t) = f_q(\boldsymbol{x}(t),t)$ over $[\tau,\infty)$ with $\boldsymbol{x}(\tau) = x$.

For any formula $\psi(u,v)$, let $\mathtt{QE}\,(\exists u,\ \psi(u,v)) \triangleq \{v \mid \exists u, \text{ s.t. } \psi(u,v) \text{ holds}\}$ denote the set of all v for which $\exists u,\ \psi(u,v)$ is true. Utilizing this notation, the state-time sets can be represented inductively.

Theorem 2. *For any $q \in Q$, suppose the solution of ODE $\dot{\boldsymbol{x}}(t) = f_q(\boldsymbol{x}(t))$ with initial x at time τ is denoted by $\Psi(\,\cdot\,; x,\tau,q)$, then the state-time sets can be inductively represented by*

$$\begin{aligned} X_q^0 = \mathtt{QE}\Big(\exists \delta \geq 0,\ &\Big(\phi_2[(x,t)=(\Psi(t+\delta;x,t,q),t+\delta)] \wedge (t+\delta \in I)\Big) \\ &\wedge \Big(\forall 0 \leq h \leq \delta,\ \phi_1[(x,t)=(\Psi(t+h;x,t,q),t+h)]\Big)\Big) \end{aligned} \tag{3}$$

$$\begin{aligned} X_q^i = \bigvee_{q'\neq q} \mathtt{QE}\Big(\exists \delta \geq 0,\ &\Big(X_{q'}^{i-1}[(x,t)=(\Psi(t+\delta;x,t,q),t+\delta)]\Big) \\ &\wedge \Big(\forall 0 \leq h \leq \delta,\ \phi_1[(x,t)=(\Psi(t+h;x,t,q),t+h)]\Big)\Big) \end{aligned} \tag{4}$$

for any $q \in Q$ and any $i \in \mathbb{N}$.

Remark 3. When $\psi(u,v)$ consists of a Boolean combination of polynomial inequalities, a decidable procedure, such as cylindrical algebraic decomposition [2], exists for computing $\mathtt{QE}\,(\exists u,\ \psi(u,v))$. This procedure exhibits a complexity that is double exponential with respect to the number of variables involved.

[1] For any $a,b \in \mathbb{R}_{>0}$ such that $a \leq b$, the constraint $a \leq t \leq b$ is concisely denoted as $t \in [a,b]$.

Remark 4. Our methodology essentially shares the idea of backward induction in controller synthesis for timed games [9]. However, our approach diverges in two key aspects: (1) the safety/target sets and timing constraints are intricately interwoven in ST-RA formula, necessitating their concurrent consideration at each step of the induction process; (2) our method operates within an infinite-dimensional space due to the continuous nature of the state space, in contrast to the backward induction for timed games, which is confined to a finite set of k-polyhedra.

4.2 Computing/Approximating State-Time Sets

Although Thm. 2 offers an inductive representation of X_q^i, the explicit computation of Eqs. (3) and (4) are challenging. This difficulty arises from two main factors: (i) the necessity to explicitly solve the ordinary differential equation in each mode, and (ii) the high complexity of QE, and the potential inclusion of non-elementary functions (such as exponential functions) in Eqs. (3) and (4), for which a generally decidable procedure to solve QE may not exist.

To address the difficulties outlined above, we categorize the dynamics into constant and non-constant systems. For the constant dynamics, its solution can be directly computed, and there exists a decidable procedure to solve QE with a complexity polynomially dependent on the formula length. For the non-constant dynamics, due to their high complexity, we forego an explicit solution for the state-time set and instead demonstrate a method to approximate this set.

Constant Dynamics. Suppose the dynamics within each mode $q \in Q$ is constant, and both ϕ_1 and ϕ_2 are Boolean combinations of linear inequalities, Eqs. (3) and (4) can be effectively solved using readily available solvers, such as Z3 [10]. Thm. 2 directly implies the following result.

Corollary 2. *Following the notations as before, suppose the dynamics within each mode is constant, i.e.* $f_q = a_q$ *for any* $q \in Q$*, and both* ϕ_1 *and* ϕ_2 *are Boolean combinations of linear inequalities, then* $\{X_q^i\}_{q \in Q}$ *can be inductively solved by Eqs.* (3) *and* (4) *with* $\Psi(t; x, \tau, q) = x + (t - \tau) \cdot a_q$.

Remark 5. Although QE on polynomial constraints is double-exponential in general [2], constant dynamics facilitate a relatively efficient (*polynomial in formula length*) solving procedure. This comes from the following observation: (1) the QE procedure in Eqs. (3) and (4) operates in polynomial time when the constraints are linear and involve only a single existential and a single universal variable [46, Thm 6.2]. (2) if X_q^{i-1} is linear for all $q \in Q$, then X_q^i is also linear.

We now illustrate the computation process of X_q^i via the following example.

Example 2. Let's reconsider the reactor system in Exmp. 1. The reactor system consists of two modes q_1 and q_2 with $f_{q_1} = 1$, $f_{q_2} = -1$, and the liquid level requirement is $\varphi = (0 \leq h \leq 4)\mathcal{U}_{[3,4]}(3 \leq h \leq 5)$.

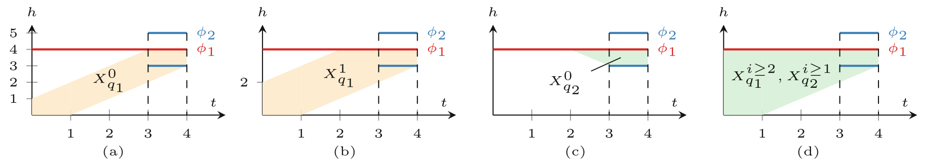


Fig. 3. The state-time sets calculation of the reactor system in Exmp. 1. The state-time sets reach a fixpoint after 2 iteration.

Based on Eqs. (3) and (4), the state-time sets we calculate are illustrated in Fig. 3. The procedure reaches a fixpoint within 2 iterations for any $q \in Q$. $\lhd$

Non-constant Dynamics. Assuming that the dynamics are non-constant, the exact computation of Eqs. (3) and (4) may prove to be overly complex or potentially undecidable. We thus seek to *inner-approximate* the state-time sets. According to Thm. 1,

- X^0_q is the set from which the system in mode q will satisfy $\phi_1 \mathcal{U} (\phi_2 \wedge (t \in I))$;
- X^i_q is the set from which the system in mode q will satisfy $\phi_1 \mathcal{U} X^{i-1}_{q'}$ for some $q' \in Q$.

We identify that the crucial element for inner-approximating the state-time sets lies in employing a method that finds sets from which the system will satisfy a classical 'until' or 'reach-avoid' formula[2]. Numerous studies have explored this issue; in this paper, we employ the approach proposed in [52].

Theorem 3 (Inner-approximation of Reach-avoid Set [52]**).** *Given dynamic system $\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t))$, safety set $\psi_1 \subseteq \mathbb{R}^n$ and target set $\psi_2 \subseteq \mathbb{R}^n$. If there exists continuously differentiable function $v(x) : \overline{\psi_1} \to \mathbb{R}$ and $w(x) : \overline{\psi_1} \to \mathbb{R}$, satisfying*[3]

$$\begin{cases} \nabla_x v(x) \cdot f(x) \geq 0, \quad \forall x \in \overline{\psi_1 \setminus \psi_2}, \\ v(x) - \nabla_x w(x) \cdot f(x) \leq 0, \quad \forall x \in \overline{\psi_1 \setminus \psi_2}, \\ v(x) \leq 0, \quad \forall x \in \partial \psi_1, \end{cases}$$

then any trajectory starting from $\{x \mid v(x) \geq 0\}$ satisfies formula $\psi_1 \mathcal{U} \psi_2$.

Remark 6. The synthesis of function $v(x)$ and $w(x)$ can be reduce to a SDP problem. For a detailed formulation, we refer the reader to [52].

Since the state-time sets X^i_q depend on both the state x and time t, we first lift the dynamics of each mode to a higher dimension that incorporates time

[2] This problem is also referred to as the inner approximation of the reach-avoid problem.

[3] $\nabla_x v(x)$ represents the gradient of $v(x)$ with respect to x, $\overline{\psi_1}$ denotes the closure of set ψ_1 and $\partial\psi_1$ refers to the boundary of ψ_1.

t. Specifically, the dynamics in mode q are transformed into $\left(\dot{x}, \dot{t}\right) = (f_q, 1)$. Subsequently, employing Thms. 3 and 1, we can inductively inner-approximate the state-time set X_q^i. The resulting approximation is denoted by $\widetilde{X_q^i}$.

Example 3. Consider a temperature control system featuring two modes, q_1 and q_2, with dynamics given by $f_{q_1} = 20 - 0.2x - 0.001x^2$ and $f_{q_2} = -0.2x - 0.001x^2$, where x represents the temperature. The control objective is defined by the ST-RA formula $\varphi = (20 \leq x \leq 80)\mathcal{U}_{[4,5]}(60 \leq x \leq 80)$.

Figure 4 presents the result obtained by inner-approximating[4] $X_{q_1}^0$, $X_{q_2}^0$, $X_{q_1}^1$, and $X_{q_2}^0$. Based on the results, we observe that when x is within the range of $[20, 80]$ in mode q_1, the system can satisfy φ without any switching. However, for $x \in [20, 80]$ in mode q_2, at least one switch is necessary for φ to be satisfied. ◁

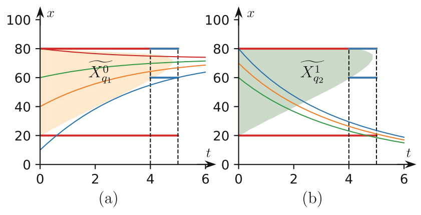


Fig. 4. The state-time sets approximation of temperature controller system

Algorithm 1. Synthesis of Switched system

```
Require: $Q$, $F$, $\varphi = \phi_1 \mathcal{U}_I \phi_2$, and $k$      ▷ $k$ is the upper bound of switching time
Ensure: A switched system $\Phi = (Q, F, \texttt{Init}, \pi)$, such that $\Phi \models \varphi$
 1: for all $q \in Q$ do
 2:     $X_q^0 \leftarrow$ inner-approximate/explicitly calculate $X_q^0$
 3:     $\texttt{Init}(q)^0 \leftarrow X_q^0[t=0]$
 4: end for
 5: for $i = 1, 2, \cdots, k$ do
 6:     for all $q \in Q$ do
 7:         $X_q^i \leftarrow$ inner-approximate/explicitly calculate $X_q^i$
 8:         $\texttt{Init}(q)^i \leftarrow (X_q^i \setminus X_q^{i-1})[t=0]$ ▷ $\texttt{Init}(q)^i$ is recorded for controller synthesis
 9:     end for
10: end for
11: $\texttt{Init} \leftarrow \cup_{q \in Q} \cup_{i=0}^{k} \texttt{Init}(q)^i$      ▷ Initial set
12: Call Alg. 2 to obtain controller $\pi$      ▷ Given any $x_0 \in \texttt{Init}$, Alg. 2 computes
                                                   the controller that drives $x_0$ to satisfy $\varphi$
```

5 Synthesizing Switched Systems

In this section, we demonstrate the synthesis of a switched system Φ that conforms to the formula $\varphi = \phi_1, \mathcal{U}_I, \phi_2$. This synthesis builds on the state-time sets introduced in Sect. 4. We initially outline the synthesis procedure for the switched system in Alg. 1 and subsequently describe the extraction of a switching controller in Alg. 2.

[4] The approximation of $X_{q_2}^0$ and $X_{q_1}^1$ is an empty set, hence it is not depicted.

Switched System Synthesis. We now summary the synthesis algorithm in Alg. 1. Given any $k \in \mathbb{N}$ that serves as a prescribed upper bound of switching time, Alg. 1 inductively calculates/inner-approximates[5] state-time sets $\{X_q^i\}_{q\in Q}$ (line 2, 7), and partition $X_q^k[t=0]$ into $\texttt{Init}(q)^i \triangleq (X_q^i \setminus X_q^{i-1})[t=0]$ (line 3, 8) for $i = 0, 1, \ldots, k$. $\texttt{Init}(q)^i$ denote the set of states (in mode q) that can be driven to satisfy φ with *at least i times* of switching (cf. Cor. 1). The initial set is defined by

$$\texttt{Init} = \cup_{q\in Q} \cup_{i=0}^{k} \texttt{Init}(q)^i,$$

which contains states that can be driven to satisfy φ within k times of switching, and the switching controller π is synthesized by Alg. 2 (line 12).

Switching Controller Synthesis. For any $x_0 \in \texttt{Init}$, Alg. 2 computes the controller that drives x_0 to satisfy φ. Alg. 2 first finds $\texttt{Init}(q_0)^l$ that contains x_0 with l be the smallest index (line 1). l is the smallest switching time that can drive x_0 to satisfy φ, and the subscript q_0 indicates x_0 first lies in mode q_0.

Algorithm 2. Switching controller synthesis

```
Require: $x_0$, $\{X_q^i\}_{i=0}^k$, and $\{\texttt{Init}(q)^i\}_{i=0}^k$          ▷ $x_0$ is the initial state
Ensure: $\pi(x_0)$                                              ▷ The switching controller
 1: Find the initial set $\texttt{Init}(q_0)^l$ that includes $x_0$ and has the smallest index $l$
 2: Select $q_0$ as initial mode, $t_0 \leftarrow 0$
 3: for $j = 1, \cdots, l$ do
 4:     for $q \in Q$ do
 5:         if $\texttt{Reach}(\widetilde{t}; x_{j-1}, t_{j-1}, q_{i-1}) \subseteq X_{\widetilde{q}}^{l-j}[t = \widetilde{t}]$ for some $\widetilde{t} > t_{j-1}, \widetilde{q} \in Q$ then
 6:             Select $t_j \leftarrow \widetilde{t}$,  $q_j \leftarrow \widetilde{q}$
 7:             $x_j \leftarrow \texttt{Reach}(t_j; x_{j-1}, t_{j-1}, q_{j-1})$
 8:             Break
 9:         end if
10:     end for
11: end for
12: $\pi(x_0) = (q_0, t_0)(q_1, t_1) \cdots (q_l, t_l)$      ▷ Representing a piecewise constant function
                                                    such that $\pi(x_0)(t) = q_i$ if $t_i \le t < t_{i+1}$
```

Line 4–10 find the next switching time and switching mode. Let $\texttt{Reach}(t; x_0, t_0, q)$ denote the over-approximation of the reachable set starting from (x_0, t_0) in mode q at time t. Next switching time $\widetilde{t}$ and switching mode $\widetilde{q}$ are chosen to ensure that the system enters $X_{\widetilde{q}}^{l-j}$ at time $\widetilde{t}$ in mode q_{j-1}, this is formally encoded by

$$\texttt{Reach}(\widetilde{t}; x_{j-1}, t_{j-1}, q_{i-1}) \subseteq X_{\widetilde{q}}^{l-j}[t = \widetilde{t}].$$

[5] To clarify, we continue to use X_q^i to represent the inner approximation of the state-time sets, rather than using $\widetilde{X_q^i}$.

In line 12, the controller π maps x_0 to a piecewise constant function $\pi(x_0) = (q_0, t_0)(q_1, t_1)\cdots(q_l, t_l)$, which represents a function that maps t to q_i if $t_i \le t < t_{i+1}$.

Remark 7. Numerous methods are available to estimate the reachable set of a dynamic system [6,50,51]. In this paper, we employ Flow* [7], a method based on Taylor model, to over-approximate the reachable set.

Remark 8. Assuming that the dynamics (i.e. f_q) within each mode remain constant, the reachable set can be explicitly calculated. This, in conjunction with the explicit calculation of state-time sets, is crucial for demonstrating relative completeness in the context of constant dynamics (c.f. Thm. 4).

Remark 9. For non-constant dynamics, since the state-time sets and reachable sets are inner- and over-approximated, there may exist an initial state x_0 that can be driven to satisfy the ST-RA formula, while our method fails to identify a controller.

We now illustrate our approach through two examples.

Example 4. In Exmp. 2, we have obtained the state-time sets $\{X^i_{q_1}, X^i_{q_2}\}$ for $i \le 2$, thus, according to Alg. 1 (with $k = 2$), we have

$$\begin{aligned} &\texttt{Init}(q_1)^0 = [0,1], && \texttt{Init}(q_1)^1 = (1,2], && \texttt{Init}(q_1)^2 = (2,4] \\ &\texttt{Init}(q_2)^0 = \emptyset, && \texttt{Init}(q_2)^1 = [0,4], && \texttt{Init}(q_2)^2 = \emptyset. \end{aligned}$$

Based on these sets, we can synthesize a switched system Φ with $\texttt{Init} = \{h \mid 0 \le h \le 4\}$. The corresponding switching controller π is defined by

$$\pi(x_0) = \begin{cases} (q_1, 0), & \text{if } 0 \le x_0 \le 1 \\ (q_2, 0)(q_1, \frac{x_0 - 1}{2}), & \text{if } 1 < x_0 \le 4. \end{cases} \qquad \lhd$$

Example 5. Let's reconsider Exmp. 3, we demonstrate our approach by synthesizing the switching controller for initial state $x_0 = 80$ in mode q_2. The reachable set $\texttt{Reach}(t; x_0, t_0, q_2)$ is represented by green boxes in Fig. 5. We observe the reachable set will enter $X^0_{q_1}$ for any $t \in [0, 2]$, this implies initial state $x_0 = 80$ in mode q_2 can be driven to satisfy φ if the system switches into mode q_1 within time interval $[0, 2]$, i.e. $\pi(80) = (q_2, 0)(q_1, \widetilde{t})$ for any $\widetilde{t} \in [0, 2]$. $\lhd$

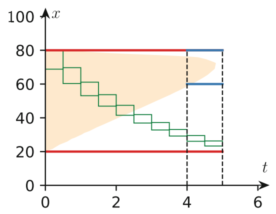


Fig. 5. Switching controller synthesis of Exmp. 3

The following result states the advantages of our approach.

Theorem 4 (Soundness, Relative Completeness, Minimal Switching Property). *Given modes Q, vector fields F, and formula $\varphi = \phi_1 \mathcal{U}_I \phi_2$, the following results hold:*

1. *Alg. 1 is sound, that is $\Phi \models \varphi$;*
2. *Alg. 1 is relatively complete for constant dynamics: for any $x \in \mathbb{R}^n$, if x can be driven to satisfy φ with some controller π, then there exists $k \in \mathbb{N}$[6], such that the initial set of the synthesized switched system contains x.*
3. *The controller synthesized in Alg. 2 features minimal switching property for constant dynamics: for any $x_0 \in$ `Init`, there does not exists any controller π', that can drive x_0 to satisfy φ with switching time (equivalently, number of discontinuous points of $\pi'(x_0)$) less than $\pi(x_0)$.*

Remark 10. Suppose the dynamic in each mode can be explicitly solved and there exists a decidable procedure for solving $\mathtt{QE}(\cdot)$ in Eqs. (3) and (4), then Alg. 1 is also relatively complete and the corresponding controller also features minimal switching property.

6 Experimental Evaluation

We develop a prototype of our synthesis method in Python, employing the Z3 solver [10] to explicitly compute the state-time sets for HSs with constant dynamics[7]. For HSs with linear or polynomial dynamics, we use the semidefinite programming solver MOSEK [30] to approximate the state-time set. The prototype is evaluated on various benchmark examples using a laptop with a 3.49 GHz Apple M2 processor, 8 GB RAM, and macOS 14.3.

As shown in Table 1, our experiments involve five distinct models, with three exhibiting constant dynamics and two exhibiting non-constant dynamics. We adjust the model scale or the ST-RA formula for each model to assess the efficacy of our method under varying conditions. And there are 15 different benchmarks in total included in our study. Table 2 details the empirical results of the benchmarks. In each case, the synthesis process continues iterating until either a fixpoint is achieved or the maximum calculation time of 5 min is met.

Our empirical results illustrate that our method is capable of effectively synthesizing controllers for models with both constant and non-constant dynamics. Notably, for models with constant dynamics, the iterative process tends to converge to a fixpoint, meaning that a complete controller is achieved. Moreover, the synthesis time for these controllers is significantly influenced by both the scale of the model and the complexity of ST-RA formulas. Specially, our analysis reveals: (i) an increased number of modes (Reactor) or a higher state dimension (CarSeq) both lead to prolonged synthesis times, (ii) more intricate predicates

[6] In fact, k can be chosen to the number of discontinuous points of $\pi(x)$.

[7] The results and the scripts to reproduce them are available at Figshare [17] or Github https://github.com/Han-SU/BenchMark_STLControlSyn4HS.

Table 1. ST-RA Specifications

Model	ST-RA Formulas
Reactor [55]	φ : $(10 \le tempe \le 90) \wedge (0 \le cooling \le 1)\, \mathcal{U}_{[15,20]}\, (40 \le tempe \le 50)$
WaterTank [33]	φ_1 : $(10 \le lev_0 \le 95) \wedge (10 \le lev_1 \le 95) \wedge (\lvert lev_0 - lev_1 \rvert \le 10)\, \mathcal{U}_{[50,60]}\, (50 \le lev_0 \le 80) \wedge (50 \le lev_1 \le 80)$
	φ_2 : $(10 \le lev_0 \le 95) \wedge (10 \le lev_1 \le 95) \wedge (\lvert lev_0 - lev_1 \rvert \le 10)\, \mathcal{U}_{[30,40]}\, (50 \le lev_0 \le 80) \wedge (50 \le lev_1 \le 80)$
	φ_3 : $(10 \le lev_0 \le 95) \wedge (10 \le lev_1 \le 95)\, \mathcal{U}_{[30,40]}\, (50 \le lev_0 \le 80) \wedge (50 \le lev_1 \le 80)$
CarSeq [5]	φ_1 : $(1 \le pos_0 - pos_1 \le 3)\, \mathcal{U}_{[2,3]}\, (20 \le pos_0 \le 25)$
	φ_2 : $(1 \le pos_0 - pos_1 \le 3) \wedge (1 \le pos_1 - pos_2)\, \mathcal{U}_{[2,3]}\, (20 \le pos_0 \le 25)$
	φ_3 : $(1 \le pos_0 - pos_1 \le 3) \wedge (1 \le pos_1 - pos_2 \le 3) \wedge (1 \le pos_2 - pos_3)\, \mathcal{U}_{[2,3]}\, (20 \le pos_0 \le 25)$
Oscillator [52]	φ : $(x^2 + y^2 \le 1)\, \mathcal{U}_{[3,4]}\, (x^2 + y^2 \le 0.01)$
Temperature [5]	φ_1 : $\wedge_{i=1,2,3}(23 \le temp_i \le 29)\, \mathcal{U}_{[8,10]} \wedge_{i=1,2,3} (26 \le temp_i \le 28)$
	φ_2 : $\wedge_{i=1,2,3}(23 \le temp_i \le 29)\, \mathcal{U}_{[8,10]} \wedge_{i=1,2,3} (26 \le temp_i \le 28) \wedge (temp_2 \le temp_1)$
	φ_3 : $\wedge_{i=1,2,3}(23 \le temp_i \le 29)\, \mathcal{U}_{[8,10]} \wedge_{i=1,2,3} (26 \le temp_i \le 28) \wedge (temp_2 \le temp_1) \wedge (temp_3 \le temp_2)$

More detail explanation of the ST-RA formula can be found in the full version of this paper [40].

Table 2. Empirical results on benchmark examples

Model	Dynamics	ST-RA	Model Scale		Synthesis Time	
			n_{dim}	n_{mode}	#Iter.	Time (s)
Reactor [55]	Const	φ	2	4	6 (fp)	0.31
		φ	2	8	6 (fp)	4.14
		φ	2	10	6 (fp)	8.01
WaterTank [33]	Const	φ_1	2	7	9 (fp)	18.04
		φ_2	2	7	6 (fp)	10.63
		φ_3	2	7	6 (fp)	5.24
CarSeq [5]	Const	φ_1	2	4	5 (fp)	1.12
		φ_2	3	8	7 (fp)	47.41
		φ_3	4	16	4	134.79
Oscillator [52]	Poly	φ	2	3	6	77.20
		φ	2	4	6	106.09
		φ	2	5	6	155.77
Temperature [5]	Linear	φ_1	3	8	5	236.99
		φ_2	3	8	5	293.66
		φ_3	3	8	5	252.32

Dynamics: the type of continuous dynamics; ST-RA: formulas to be satisfied (cf. Table 1);
n_{dim}: dimension of state; n_{mode}: number of modes; #Iter.: number of iterations, (fp) means the synthesized set X_q^i (cf. Sect. 5) reach a fixpoint at current iteration.

or larger future-reach time[8] (WaterTank) results in increased synthesis times. For the third benchmark within CarSeq, the model does not reach a fixpoint, primarily because the large model scale rapidly increase the formula size, posing substantial challenges for the Z3 solver.

When dealing with non-constant dynamics, an approximation method is applied, thereby a fixpoint might not be achievable. The influence of model scale on synthesis time remains consistent with that observed in constant ODE models, as evidenced in Oscillator. Interestingly, the synthesis time for controllers using approximation methods is less affected by the complexity of the ST-RA formula. For example, in Temperature, despite φ_3 being more complex than φ_2, it requires less synthesis time, primarily because the complexity of SDP is influenced more by state space dimensions than by constraints.

Overall, our method exhibits a high capability in synthesizing switching controller for HSs with various dynamics. It can achieve sound and complete results for constant dynamics within a reasonable time. For more general dynamics, our method can still synthesize a sound result in a reasonable time.

7 Related Work

HSs have been a key research focus in the academic community [47]. The autonomous *verification* and *synthesis* of HSs began from timed automata [1]. Subsequently, various mathematical models, including hybrid automata [18,19], delay differential systems [14,56], stochastic systems [13,31], have been employed to reason about HSs. For a survey of these methods, we refer to [11].

In the realm of formal synthesis of HSs, different methods [21,42] can be classified along several dimensions. (i) Along the designable part of the system, the synthesis problem can be categorized into *feedback controller* synthesis [38], *switching controller* synthesis [20,22], and *reset controller* synthesis [8,25,41]. (ii) Along the properties of interest, the problem can be classified into *safety* controller synthesis, *liveness* controller synthesis, etc.

Switching controller synthesis [22], shaping HSs by strategically constraining their discrete behavior, can be categorized into two fundamentally approaches. The first is based on constraint solving [42,55]. This approach highly dependents on finding suitable certificate templates, which is challenging to generate manually. The other approach is abstraction-based method. Given its capability to easily handle complex temporal specifications, this method has been increasingly adopted in recent research [4,16,26].

The synthesis of HSs against reach-avoid type specifications, similar to those discussed in this paper, predominantly focuses on feedback controllers. Notable methods include Counterexample-Guided Inductive Synthesis (CEGIS) [36,37], optimization-based methods [52], and others [15,32].

[8] Future-reach time represents the maximum time horizon required to verify the correctness of an STL formula [5], in WaterTank, the future-reach times of φ_1, φ_2, and φ_3 are 60, 40, and 40 respectively.

When considering STL as the specification, most works focus solely on the continuous dynamics of HSs. Raman et al. proposed a method to encode the STL specification of a hybrid system into MILP [34]. This method was further employed to synthesize robust controllers [35]. The Control Barrier Function-based method can also extends to synthesize feedback controllers with respect to STL specification [23]. Recently, [29] utilizes reinforcement learning technique to synthesize robust controllers for essential discrete-time systems. To the best of our knowledge, [39] is the only work that considers controller synthesis problem of switched systems against STL specification. However, [39] focuses on synthesizing switch input for the hybrid automata with discrete dynamics, while our work aims at synthesizing switching controllers that determine the switch time for the system.

8 Conclusion

We proposed a novel method to synthesize switching controllers for HSs against a fragment of the STL. Our method iteratively calculates the state-time set for each mode, which services as foundation of the synthesize algorithm. The distinctive feature of our approach lies in its soundness and relative completeness. Our preliminary experiments, leveraging a range of notable examples from existing literature, have effectively demonstrated the method's efficiency and efficacy.

For future work, we plan to continue to explore in two directions. (i) Enlarge the range of specifications under consideration to encompass general STL formulas featuring nested temporal operators. The primary challenge here is devising a unified, recursive formula reasoning approach for general STL specifications. (ii) Broaden the types of controllers that can be synthesized from the calculated state-time sets.

Data Availability Statement. The experimental results of this paper may be reproduced using the artifact on Figshare [17].

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoret. Comput. Sci. **126**(2), 183–235 (1994)
2. Arnon, D.S., Collins, G.E., McCallum, S.: Cylindrical algebraic decomposition i: the basic algorithm. SIAM J. Comput. **13**(4), 865–877 (1984)
3. Atkins, E.M., Bradley, J.M.: Aerospace cyber-physical systems education. In: AIAA Infotech@ Aerospace (I@ A) Conference, p. 4809 (2013)
4. Aydin Gol, E., Lazar, M., Belta, C.: Language-guided controller synthesis for discrete-time linear systems. In: Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control, pp. 95–104 (2012)
5. Bae, K., Lee, J.: Bounded model checking of signal temporal logic properties using syntactic separation. Proc. ACM Program. Lang. **3**(POPL), 1–30 (2019)
6. Chen, X., Abraham, E., Sankaranarayanan, S.: Taylor model flowpipe construction for non-linear hybrid systems. In: 2012 IEEE 33rd Real-Time Systems Symposium, pp. 183–192. IEEE (2012)

7. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
8. Clegg, J.C.: A nonlinear integrator for servomechanisms. Trans. Am. Inst. Electr. Eng. Part II Appl. Ind. **77**(1), 41–42 (1958)
9. de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: The element of surprise in timed games. In: Amadio, R., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 144–158. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45187-7_9
10. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
11. Deshmukh, J.V., Sankaranarayanan, S.: Formal techniques for verification and testing of cyber-physical systems. In: Al Faruque, M.A., Canedo, A. (eds.) Design Automation of Cyber-Physical Systems, pp. 69–105. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-13050-3_4
12. Engell, S., Kowalewski, S., Schulz, C., Stursberg, O.: Continuous-discrete interactions in chemical processing plants. Proc. IEEE **88**(7), 1050–1068 (2000)
13. Feng, S., Chen, M., Xue, B., Sankaranarayanan, S., Zhan, N.: Unbounded-time safety verification of stochastic differential dynamics. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 327–348. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_18
14. Feng, S., Chen, M., Zhan, N., Fränzle, M., Xue, B.: Taming delays in dynamical systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 650–669. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_37
15. Fränzle, M., Chen, M., Kröger, P.: In memory of oded maler: automatic reachability analysis of hybrid-state automata. ACM SIGLOG News **6**(1), 19–39 (2019)
16. Girard, A.: Controller synthesis for safety and reachability via approximate bisimulation. Automatica **48**(5), 947–953 (2012)
17. Han, S., Shenghua, F., Sinong, Z., Naijun, Z.: Benckmark examples of paper "switching controller synthesis for hybrid systems against STL formulas." Figshare. Software (2024). https://doi.org/10.6084/m9.figshare.26057410.v1
18. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings 11th Annual IEEE Symposium on Logic in Computer Science, pp. 278–292. IEEE (1996)
19. Henzinger, T.A., Majumdar, R.: Symbolic model checking for rectangular hybrid systems. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 142–156. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46419-0_11
20. Jha, S., Seshia, S.A., Tiwari, A.: Synthesis of optimal switching logic for hybrid systems. In: Proceedings of the Ninth ACM International Conference on Embedded Software, pp. 107–116 (2011)
21. Jin, X., An, J., Zhan, B., Zhan, N., Zhang, M.: Inferring switched nonlinear dynamical systems. Formal Aspects Comput. **33**(3), 385–406 (2021)
22. Liberzon, D.: Switching in Systems and Control, vol. 190. Springer (2003). https://doi.org/10.1007/978-1-4612-0017-8
23. Lindemann, L., Dimarogonas, D.V.: Control barrier functions for signal temporal logic tasks. IEEE Control Syst. Lett. **3**(1), 96–101 (2018)
24. Lindemann, L., Nowak, J., Schönbächler, L., Guo, M., Tumova, J., Dimarogonas, D.V.: Coupled multi-robot systems under linear temporal logic and signal temporal logic tasks. IEEE Trans. Control Syst. Technol. **29**(2), 858–865 (2019)

25. Liu, J., et al.: Correct-by-construction for hybrid systems by synthesizing reset controller. arXiv preprint arXiv:2309.05906 (2023)
26. Liu, J., Ozay, N., Topcu, U., Murray, R.M.: Synthesis of reactive switching protocols from temporal logic specifications. IEEE Trans. Autom. Control **58**(7), 1771–1785 (2013)
27. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
28. Mazo, M., Davitian, A., Tabuada, P.: PESSOA: a tool for embedded controller synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 566–569. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_49
29. Meng, Y., Fan, C.: Signal temporal logic neural predictive control. IEEE Robot. Autom. Lett. **8**(11), 7719–7726 (2023). https://doi.org/10.1109/LRA.2023.3315536
30. Mosek, A.: The MOSEK optimization toolbox for MATLAB manual. Version 7.1 (revision 28) (2015). http://mosek.com. Accessed 20 Mar 2015
31. Prajna, S., Jadbabaie, A., Pappas, G.J.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE Trans. Autom. Control **52**(8), 1415–1428 (2007)
32. Prajna, S., Rantzer, A.: Convex programs for temporal verification of nonlinear dynamical systems. SIAM J. Control. Optim. **46**(3), 999–1021 (2007)
33. Raisch, J., Klein, E., Meder, C., Itigin, A., O'Young, S.: Approximating automata and discrete control for continuous systems — two examples from process control. In: Antsaklis, P., Lemmon, M., Kohn, W., Nerode, A., Sastry, S. (eds.) HS 1997. LNCS, vol. 1567, pp. 279–303. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49163-5_16
34. Raman, V., Donzé, A., Maasoumy, M., Murray, R.M., Sangiovanni-Vincentelli, A., Seshia, S.A.: Model predictive control with signal temporal logic specifications. In: 53rd IEEE Conference on Decision and Control, pp. 81–87. IEEE (2014)
35. Raman, V., Donzé, A., Sadigh, D., Murray, R.M., Seshia, S.A.: Reactive synthesis from signal temporal logic specifications. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, pp. 239–248 (2015)
36. Ravanbakhsh, H., Sankaranarayanan, S.: Counterexample-guided stabilization of switched systems using control lyapunov functions. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, pp. 297–298 (2015)
37. Ravanbakhsh, H., Sankaranarayanan, S.: Robust controller synthesis of switched systems using counterexample guided framework. In: Proceedings of the 13th International Conference on Embedded Software, pp. 1–10 (2016)
38. Sanfelice, R.G.: Hybrid Feedback Control. Princeton University Press (2021)
39. da Silva, R.R., Kurtz, V., Lin, H.: Symbolic control of hybrid systems from signal temporal logic specifications. Guidance Navig. Control **1**(02), 2150008 (2021)
40. Su, H., Feng, S., Zhan, S., Zhan, N.: Switching controller synthesis for hybrid systems against STL formulas. arXiv preprint arXiv:2406.16588 (2024)
41. Su, H., et al.: Reset controller synthesis by reach-avoid analysis for delay hybrid systems. arXiv preprint arXiv:2309.05908 (2023)
42. Taly, A., Gulwani, S., Tiwari, A.: Synthesizing switching logic using constraint solving. Int. J. Softw. Tools Technol. Transfer **13**(6), 519–535 (2011)

43. Tomlin, C.J., Lygeros, J., Sastry, S.S.: A game theoretic approach to controller design for hybrid systems. Proc. IEEE **88**(7), 949–970 (2000)
44. Wang, Y., et al.: Joint differentiable optimization and verification for certified reinforcement learning. In: Proceedings of the ACM/IEEE 14th International Conference on Cyber-Physical Systems (with CPS-IoT Week 2023), pp. 132–141 (2023)
45. Wang, Y., et al.: Enforcing hard constraints with soft barriers: safe reinforcement learning in unknown stochastic environments. In: International Conference on Machine Learning, pp. 36593–36604. PMLR (2023)
46. Weispfenning, V.: The complexity of linear problems in fields. J. Symb. Comput. **5**(1–2), 3–27 (1988)
47. Witsenhausen, H.: A class of hybrid-state continuous-time dynamic systems. IEEE Trans. Autom. Control **11**(2), 161–167 (1966)
48. Wu, Q., et al.: Boosting long-delayed reinforcement learning with auxiliary short-delayed task. arXiv preprint arXiv:2402.03141 (2024)
49. Wu, Q., et al.: Variational delayed policy optimization. arXiv preprint arXiv:2405.14226 (2024)
50. Xue, B., Fränzle, M., Zhan, N.: Inner-approximating reachable sets for polynomial systems with time-varying uncertainties. IEEE Trans. Autom. Control **65**(4), 1468–1483 (2019)
51. Xue, B., She, Z., Easwaran, A.: Under-approximating backward reachable sets by polytopes. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 457–476. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_25
52. Xue, B., Zhan, N., Fränzle, M., Wang, J., Liu, W.: Reach-avoid verification based on convex optimization. IEEE Trans. Autom. Control **69**, 598–605 (2023)
53. Ye, P., Entcheva, E., Smolka, S.A., Grosu, R.: Modelling excitable cells using cycle-linear hybrid automata. IET Syst. Biol. **2**(1), 24–32 (2008)
54. Zhan, S.S., Wang, Y., Wu, Q., Jiao, R., Huang, C., Zhu, Q.: State-wise safe reinforcement learning with pixel observations. arXiv preprint arXiv:2311.02227 (2023)
55. Zhao, H., Zhan, N., Kapur, D.: Synthesizing switching controllers for hybrid systems by generating invariants. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 354–373. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39698-4_22
56. Zou, L., Fränzle, M., Zhan, N., Mosaad, P.N.: Automatic verification of stability and safety for delay differential equations. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 338–355. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_20

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

On Completeness of SDP-Based Barrier Certificate Synthesis over Unbounded Domains

Hao Wu[1], Shenghua Feng[2], Ting Gan[3], Jie Wang[4], Bican Xia[5], and Naijun Zhan[1,6](✉)

[1] State Key Laboratory of Computer Science, Institute of Software, University of Chinese Academy of Sciences, Beijing, China
{wuhao,znj}@ios.ac.cn

[2] Zhongguancun Laboratory, Beijing, China
fengsh@zgclab.edu.cn

[3] School of Computer Science, Wuhan University, Wuhan, China
ganting@whu.edu.cn

[4] Academy of Mathematics and Systems Science, CAS, Beijing, China
wangjie212@amss.ac.cn

[5] School of Mathematical Sciences, Peking University, Beijing, China
xbc@math.pku.edu.cn

[6] School of Computer Science, Peking University, Beijing, China

Abstract. Barrier certificates, serving as differential invariants that witness system safety, play a crucial role in the verification of cyber-physical systems (CPS). Prevailing computational methods for synthesizing barrier certificates are based on semidefinite programming (SDP) by exploiting *Putinar Positivstellensatz*. Consequently, these approaches are limited by the *Archimedean condition*, which requires all variables to be bounded, i.e., systems are defined over bounded domains. For systems over unbounded domains, unfortunately, existing methods become incomplete and may fail to identify potential barrier certificates.

In this paper, we address this limitation for the unbounded cases. We first give a complete characterization of polynomial barrier certificates by using *homogenization*, a recent technique in the optimization community to reduce an unbounded optimization problem to a bounded one. Furthermore, motivated by this formulation, we introduce the definition of *homogenized systems* and propose a complete characterization of a family of non-polynomial barrier certificates with more expressive power. Experimental results demonstrate that our two approaches are more effective while maintaining a comparable level of efficiency.

Keywords: Safety · Barrier certificates · Semidefinite programming · Homogenization

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 248–266, 2025.
https://doi.org/10.1007/978-3-031-71177-0_16

1 Introduction

With recent advancements in optimization theory and computational techniques, Cyber-Physical Systems (CPS), which involve the seamless integration of physical components and software systems, have proliferated across various application domains. A significant subset of CPS, known as safety-critical systems, presents a heightened level of concern. Failures or malfunctions in such systems can lead to severe safety risks for individuals and the environment. Examples of safety-critical CPS include aircraft, automobiles, integrated medical devices, nuclear power plants, and biological systems. As a result, ensuring the safety of these systems has become a primary focus of extensive academic research.

One of the key challenges in CPS verification is the safety problem (or dually, the reachability problem), i.e., to demonstrate that a system, starting from its initial states, never enters an unsafe region. In general, the safety problem of CPS is undecidable [16]. The most challenging aspect of such problem lies in reasoning about the continuous dynamics, which are typically described by ordinary differential equations (ODEs).

Deductive verification, derived from Hoare-style program verification [17], offers a method to verify safety without directly computing the reachable set. At the core of deductive verification lies the synthesis of *differential invariants* [24,28], which extend the concept of inductive invariants to the continuous-time domain. Specifically, a differential invariant is a set of states from which any trajectories starting from it can never escape. With a priori specified template, the invariant generation problem boils down to solving the constraints encoding the invariant condition. When all involved constraints are polynomial, the problem is decidable but has time complexity doubly exponential in the number of variables [24], according to Tarski's theorem [39] and the complexity for the quantifier elimination procedure [9]. Consequently, considerable efforts have been dedicated to identifying differential invariants that allow for efficient synthesis.

In their seminal work [29], Prajna and Jadbabaie introduced the concept of barrier certificates as witnesses to safety. Namely, a barrier certificate is a real-valued function whose zero sub-level set serves as a differential invariant, separating the set of initial states and the unsafe region. It is important to note that, for the purpose of efficient synthesis, the barrier certificate condition strengthens the general condition of differential invariants. Since then, various definitions of barrier certificates have been proposed, aiming to relax the original barrier certificate conditions while still allowing for efficient synthesis. Examples of such definitions include exponential-type barrier certificates [21], Darboux-type barrier certificates [45], general convex barrier certificates [8] and vector barrier certificates [37], and invariant barrier certificates [41]. Moreover, similar notions of barrier certificates have been developed for verification problems that involve control inputs [2,44], disturbances [42], stochastic dynamics [11,18,20,30], and temporal logic specifications [25,43]. These extensions broaden the applicability of barrier certificates in various domains. Recently, there are also works aim at generalizing the notion of k-inductiveness for safety verification, leading to the definitions of t-barrier certificates [6] and k-inductive barrier certificates [3,4].

Sum-of-squares optimization is a well-established computational technique for synthesizing barrier certificates and has been employed in most of the works mentioned above. Typically, the barrier certificate conditions are first encoded into constraints involving sum-of-squares polynomials. These constraints are then translated into SDP and solved by numerical solvers. In scenarios where the domains are bounded, one can choose to rely on either a sound characterization or a complete characterization to encode the conditions. The differences between these two characterizations are often overlooked, as their formulations are quite similar. However, when dealing with systems defined over unbounded domains, the sound characterization tends to be conservative while the complete characterization can not be utilized due to the violation of the Archimedean condition in Putinar's Positivstellensatz. In such unbounded cases, existing methods solely rely on the sound characterization, potentially leading to conservative results.

Besides sum-of-squares optimization, much effort have been devoted to incorporate other numerical methods for solving the obtained constraints, for instance, interval arithmetic [10,12,13], linear programming [35], and data-driven approaches [1,27,33,46,47].

Contributions. Our main contributions are threefold:

1. We explicitly clarify the connection between the soundness and the completeness of the sum-of-squares characterization of barrier certificates, which is mostly overlooked in existing works. This can be considered as a minor contribution. (See Sect. 3)
2. We utilize the *homogenization* technique from [19] to derive the first complete sum-of-squares characterization of polynomial barrier certificates over unbounded domains. (See Sect. 4)
3. We introduce the definition of homogenized systems and consider a specific class of non-polynomial barrier certificates with more expressive power. We also propose a complete sum-of-squares characterization for this class of non-polynomial barrier certificates. (See Sect. 5)

Finally, we implement algorithms for synthesizing barrier certificates based on the existing incomplete characterization and our two novel complete characterizations. These algorithms are tested over a set of benchmarks with unbounded domains adapted from the literature. Experimental results demonstrate that the two complete characterizations are more expressive while maintaining a comparable level of efficiency. (See Sect. 6)

Organization. The rest of this paper is organized as follows: Sect. 2 introduces algebraic tools that will be used. Section 3 formulates the barrier certificate synthesis problems and explains the connection between the sound and the complete characterization in the bounded case. Section 4 proposes the first complete characterization of polynomial barrier certificates over unbounded domains. Section 5 introduces the definition of homogenized systems and extends the results to a class of non-polynomial barrier certificates. Finally, Sect. 6 reports the experimental results and Sect. 7 concludes the paper.

2 Preliminaries

In this section, we fix basic notations and introduce necessary concepts concerning sum-of-squares optimization. For interested readers, we recommend [7,23] for a detailed treatment of this topic.

Basic Notations. Let $\mathbb{N}$, $\mathbb{R}$, $\mathbb{R}_{\geq 0}$, and $\mathbb{R}_{>0}$ denote the set of all natural numbers, the set of reals, non-negative real numbers and the set of positive real numbers, respectively. The set of continuously differentiable functions over $\mathbb{R}^n$ is denoted by $\mathcal{C}^1(\mathbb{R}^n)$. By convention, we use boldface letters to denote vectors and vector-valued functions, e.g., $\boldsymbol{x} = (x_1, \ldots, x_n)$ denotes a state variable and $\boldsymbol{f} = (f_1, \ldots, f_n)$ denotes a vector field. For vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$, $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \ \hat{=}\ \sum_{i=1}^{n} x_i y_i$ represents the inner product of $\boldsymbol{x}$ and $\boldsymbol{y}$, and $\|\boldsymbol{x}\| \ \hat{=}\ \sqrt{\langle \boldsymbol{x}, \boldsymbol{x} \rangle}$ denotes the standard Euclidean norm.

Let $\mathbb{R}[\boldsymbol{x}]$ denote the set of polynomials in variables $\boldsymbol{x}$ with real coefficients. A basic semialgebraic set $\mathcal{K} \subseteq \mathbb{R}^n$ is of the form $\{\boldsymbol{x} \in \mathbb{R}^n \mid p_1(\boldsymbol{x}) \triangleright 0, \ldots, p_m(\boldsymbol{x}) \triangleright 0\}$, where $p_i(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ and $\triangleright \in \{\geq, >\}$. An equality $p(\boldsymbol{x}) = 0$ can be represented by two inequalities $p(\boldsymbol{x}) \geq 0$ and $-p(\boldsymbol{x}) \geq 0$. A basic semialgebraic set is considered *closed* when its defining polynomials contain only non-strict inequalities. Semialgebraic sets are formed as unions of basic semialgebraic sets. i.e., $\bigcup_{i=1}^{n} \mathcal{K}_i$, where each $\mathcal{K}_i$ is a basic semialgebraic set.

Sum-of-Squares Polynomials. Given $S \subseteq \mathbb{R}^n$, we say $p(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ is *nonnegative* (resp. *strictly positive*) over S if $p(\boldsymbol{x}) \geq 0$ (resp. $p(\boldsymbol{x}) > 0$) for any $\boldsymbol{x} \in S$. Sum-of-squares (SOS) polynomials are an important subset of globally nonnegative polynomials over $\mathbb{R}^n$. A polynomial $p(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ is said to be a *sum-of-squares* polynomial if it can be expressed as $p(\boldsymbol{x}) = \sum_{i=1}^{m} p_i(\boldsymbol{x})^2$, where $p_i(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ for each i. We use $\Sigma[\boldsymbol{x}]$ to denote the set of SOS polynomials in variables $\boldsymbol{x}$.

Putinar's Theorem. Given polynomials $p_1, \ldots, p_m \in \mathbb{R}[\boldsymbol{x}]$. Let $\mathcal{K}$ be a closed basic semialgebraic set described by

$$\mathcal{K} \ \hat{=}\ \{\boldsymbol{x} \in \mathbb{R}^n \mid p_1(\boldsymbol{x}) \geq 0, \ldots, p_m(\boldsymbol{x}) \geq 0\}. \tag{1}$$

The set of polynomials

$$\mathbf{QM}(p_1, p_2, \ldots, p_m) \ \hat{=}\ \left\{\sigma_0 + \sum_{i=1}^{m} \sigma_i p_i \mid \sigma_i \in \Sigma[\boldsymbol{x}] \text{ for } i = 0, 1, \ldots, m\right\}$$

is called the *quadratic module* generated by $p_1, \ldots, p_m$. A quadratic module $\mathbf{QM}$ is *Archimedean*, or satisfies the *Archimedean condition*, if $N - \|\boldsymbol{x}\|^2 \in \mathbf{QM}$ for some constant $N \in \mathbb{N}$. Since a sum-of-squares polynomial $\sigma(\boldsymbol{x}) \in \Sigma[\boldsymbol{x}]$ is nonnegative over $\mathbb{R}^n$, the following result trivially holds.

Proposition 1. *Given $\mathcal{K}$ as defined in Eq. (1), then*

$$f(\boldsymbol{x}) \in \boldsymbol{QM}(p_1, \ldots, p_m) \implies f(\boldsymbol{x}) \geq 0 \text{ over } \mathcal{K}.$$

An important result in real algebraic geometry is Putinar's Positivstellensatz, which states that, under the Archimedean condition, the quadratic module $\mathbf{QM}(p_1, \ldots, p_m)$ contains all polynomials strictly positive over $\mathcal{K}$.

Theorem 1 (Putinar's Positivstellensatz [23,31]). *Given $\mathcal{K}$ as defined in Eq. (1) and a polynomial $f \in \mathbb{R}[\boldsymbol{x}]$, if $\boldsymbol{QM}(p_1, \ldots, p_m)$ is Archimedean, then*

$$f(\boldsymbol{x}) > 0 \text{ over } \mathcal{K} \implies f(\boldsymbol{x}) \in \boldsymbol{QM}(p_1, \ldots, p_m).$$

Here, the condition "$\mathbf{QM}(p_1, \ldots, p_m)$ is Archimedean" can be intuitively understood as $\mathcal{K}$ in Eq. (1) is bounded. In one direction, if $\mathbf{QM}(p_1, \ldots, p_m)$ is Archimedean, by Proposition 1, we have $N - \|\boldsymbol{x}\|^2 \geq 0$ over $\mathcal{K}$, hence $\mathcal{K}$ is bounded. In the other direction, when $\mathcal{K}$ is bounded within a ball $\{\boldsymbol{x} \in \mathbb{R}^n \mid N - \|\boldsymbol{x}\|^2 \geq 0\}$, then we can assume a redundant constraint $p_{m+1} = N - \|\boldsymbol{x}\|^2$ and the new quadratic module $\mathbf{QM}(p_1, \ldots, p_m, p_{m+1})$ is Archimedean. In general, note that Proposition 1 does not necessarily imply that $\mathcal{K}$ is bounded.

3 Problem Formulation

In this section, we formally define the barrier certificate synthesis problem of interest, and discuss the relation between the sound and the complete sum-of-squares characterization of polynomial barrier certificates over bounded domains. The majority of the existing literature, such as [8,21,29,37], primarily focus on the sound characterization. As far as we are aware, the complete characterization is only mentioned in [41]. Subsequently, we clarify the connection between these two characterizations, which can be considered as a minor contribution.

Differential Dynamical Systems. We consider a class of dynamical systems featuring differential dynamics governed by ordinary differential equations (ODEs) of autonomous type:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) \tag{2}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ is the state vector, $\dot{\boldsymbol{x}}$ denotes its temporal derivative dx/dt, and $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$ is a polynomial vector field, i.e., each component f_i of $\boldsymbol{f}$ is a polynomial. Since a polynomial vector field is locally Lipschitz continuous, ODE (2) admits an unique *solution* (or *trajectory*), denoted as $\xi_{\boldsymbol{x}_0} : \mathbb{R}_{\geq 0} \to \mathbb{R}^n$, from any initial state $\boldsymbol{x}_0 \in \mathbb{R}^n$, such that (1) $\xi_{\boldsymbol{x}_0}(0) = \boldsymbol{x}_0$ (2) for any $t' \in \mathbb{R}_{\geq 0}$, $\frac{\mathrm{d}\xi_{x_0}}{\mathrm{d}t}\big|_{t=t'} = \boldsymbol{f}(\xi_{\boldsymbol{x}_0}(t'))$.

Safety Verification Problems. Given dynamical system Eq. (2) with domain $\mathcal{X} \subseteq \mathbb{R}^n$, initial set $\mathcal{I} \subset \mathcal{X}$, and unsafe set $\mathcal{U} \subset \mathcal{X}$, the *safety verification problem* asks whether $\mathcal{U}$ is reachable from any state in $\mathcal{I}$ within $\mathcal{X}$. Formally, let $\mathcal{R}$ denote the reachable set,

$$\mathcal{R} \ \hat{=} \ \{\boldsymbol{x} \in \mathcal{X} \mid \exists t \in \mathbb{R}_{\geq 0}, \exists \boldsymbol{x}_0 \in \mathcal{I}.\ \boldsymbol{x} = \xi_{\boldsymbol{x}_0}(t)\},$$

where we assume that a trajectory will never leave the domain. The system is said to be *safe* if $\mathcal{U} \cap \mathcal{R} = \emptyset$, and *unsafe* otherwise.

In this paper, we restrict our focus to the case when $\mathcal{X}$, $\mathcal{I}$, and $\mathcal{U}$ are closed basic semialgebraic sets described by

$$\begin{aligned}
\mathcal{X} &\hat{=} \left\{\boldsymbol{x} \in \mathbb{R}^n \mid g_i^{\mathcal{X}}(\boldsymbol{x}) \geq 0, \text{ for } i = 1, \ldots, m_x\right\}, \\
\mathcal{I} &\hat{=} \left\{\boldsymbol{x} \in \mathcal{X} \mid g_i^{\mathcal{I}}(\boldsymbol{x}) \geq 0, \text{ for } i = 1, \ldots, m_i\right\}, \\
\mathcal{U} &\hat{=} \left\{\boldsymbol{x} \in \mathcal{X} \mid g_i^{\mathcal{U}}(\boldsymbol{x}) \geq 0, \text{ for } i = 1, \ldots, m_u\right\}.
\end{aligned}$$

Invariants. A *differential invariant* is a subset $\Phi \subseteq \mathcal{X}$ such that any trajectory starting from Φ stays within Φ forever, i.e.,

$$\forall \boldsymbol{x}_0 \in \Phi, \forall t \in \mathbb{R}_{\geq 0}.\ \xi_{\boldsymbol{x}_0}(t) \in \Phi.$$

Utilizing this concept, we can verify the safety of a system without explicitly computing the reachable set, which is typically intractable for the majority of nonlinear systems. The idea therein is to find a differential invariant $\Phi \subseteq \mathcal{X}$ such that $\mathcal{I} \subseteq \Phi$ and $\mathcal{U} \subseteq \mathcal{X} \backslash \Phi$. According to the definition, the differential invariant Φ serves as an over-approximation of the reachable set $\mathcal{R}$, thereby substantiating safety of the system.

Barrier Certificates. Barrier certificates encapsulate the conditions requisite for a zero sub-level set of the form $\{\boldsymbol{x} \in \mathbb{R}^n \mid B(\boldsymbol{x}) \leq 0\}$ to become a differential invariant, where $B \in \mathcal{C}^1(\mathbb{R}^n)$. For the ease of explanation, we focus on exponential-type barrier certificates and refer to them as barrier certificates for simplicity. The technique presented in this paper can be readily extended to other types of barrier certificates [8,37,41] and hybrid systems (systems containing discrete transitions and continuous evolution) [29].

Theorem 2 (Exponential-type Barrier Certificates, Modified from [21]). *Given the system (2) with sets $\mathcal{X}$, $\mathcal{I}$, and $\mathcal{U}$. For any $\lambda \in \mathbb{R}$, the system is safe if there exists an exponential-type barrier certificate, namely a real-valued function $B(\boldsymbol{x}) \in \mathcal{C}^1(\mathbb{R}^n)$ satisfying the following conditions*

$$\forall \boldsymbol{x} \in \mathcal{I}.\ B(\boldsymbol{x}) \leq 0, \tag{3}$$

$$\forall \boldsymbol{x} \in \mathcal{U}.\ B(\boldsymbol{x}) \geq \epsilon_e, \tag{4}$$

$$\forall \boldsymbol{x} \in \mathcal{X}.\ \mathfrak{L}_{\boldsymbol{f}} B(\boldsymbol{x}) - \lambda B(\boldsymbol{x}) \leq 0, \tag{5}$$

for some real constant $\epsilon_e \in \mathbb{R}_{>0}$, where $\mathfrak{L}_{\boldsymbol{f}} p(\boldsymbol{x}) \hat{=} \langle \frac{\partial}{\partial \boldsymbol{x}} p(\boldsymbol{x}), \boldsymbol{f}(\boldsymbol{x}) \rangle$ is the Lie derivative of $p(\boldsymbol{x})$ over the vector filed $\boldsymbol{f}$.

The difference between our Theorem 2 and its original formulation in [21] lies in Eq. (4), which was written as

$$\forall \boldsymbol{x} \in \mathcal{U}.\ B(\boldsymbol{x}) > 0. \tag{4'}$$

When the unsafe region $\mathcal{U}$ is bounded (compact), the two condition Eq. (4) and Eq. (4') coincide, as a continuous function over a compact set always attains a minimum. However, when $\mathcal{U}$ is unbounded, our formulation is stricter in the sense that $\mathcal{I}$ and $\mathcal{U}$ can not be arbitrarily close, otherwise we would be unable to distinguish between them, as shown in the following Exmp. 1. In theory, ϵ_e can be any real constant in $\mathbb{R}_{>0}$, and the corresponding $B(\boldsymbol{x})$ will be equivalent up to a constant factor.

Example 1. Consider a system $\boldsymbol{f}(x_1, x_2) = (x_1, 0)$ with $\mathcal{X} = \mathbb{R}^2$, $\mathcal{I} = \{(x_1, x_2) \mid x_1x_2 + 1 \leq 0, x_1 \leq 0\}$, and $\mathcal{U} = \{(x_1, x_2) \mid x_1x_2 - 1 \geq 0, x_1 \geq 0\}$. The function $B(x_1, x_2) = x_1$ is not a valid barrier certificate according to our definition, as the condition Eq. (4) is not satisfiable for any $\epsilon_e > 0$ (though when $\epsilon_e = 0$ Eq. (4') is satisfied). In other words, the sets $\mathcal{I}$ and $\mathcal{U}$ are indistinguishable in practice when x_2 goes to infinity, and our Theorem 2 rules out such cases.

To ensure computational tractability, the barrier certificate $B(\boldsymbol{x})$ is commonly constrained to polynomial forms. One of the prevailing computational methods for synthesizing $B(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ is based on the sum-of-squares optimization. Now we present the sound and complete sum-of-squares characterizations of polynomial barrier certificate over bounded domains.

Theorem 3 (Bounded Case). *Let $\mathcal{X}$, $\mathcal{I}$, and $\mathcal{U}$ be bounded, i.e., the corresponding quadratic module is Archimedean. Given $\lambda \in \mathbb{R}$ and $\epsilon_e \in \mathbb{R}_{>0}$, consider the following constraints with parameter ϵ,*

$$\begin{aligned}
&-B(\boldsymbol{x}) + \epsilon = \sigma_0^{\mathcal{I}} + \sum_{i=1}^{m_i} g_i^{\mathcal{I}}(\boldsymbol{x})\sigma_i^{\mathcal{I}} \\
&B(\boldsymbol{x}) - \epsilon_e + \epsilon = \sigma_0^{\mathcal{U}} + \sum_{i=1}^{m_u} g_i^{\mathcal{U}}(\boldsymbol{x})\sigma_i^{\mathcal{U}} \\
&\lambda B(\boldsymbol{x}) - \mathfrak{L}_{\boldsymbol{f}} B(\boldsymbol{x}) + \epsilon = \sigma_0^{\mathcal{X}} + \sum_{i=1}^{m_x} g_i^{\mathcal{X}}(\boldsymbol{x})\sigma_i^{\mathcal{X}} \\
&\sigma_0^{\mathcal{I}}, \ldots, \sigma_{m_i}^{\mathcal{I}}, \sigma_0^{\mathcal{U}}, \ldots, \sigma_{m_u}^{\mathcal{U}}, \sigma_0^{\mathcal{X}}, \ldots, \sigma_{m_x}^{\mathcal{X}} \in \Sigma[\boldsymbol{x}].
\end{aligned} \tag{6}$$

When $\epsilon = 0$, Eq. (6) gives a sound characterization of polynomial barrier certificates, i.e., any solution $B(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ to the above constraints is a barrier certificate. When $\epsilon > 0$, Eq. (6) gives a complete characterization of polynomial barrier certificates, i.e., any barrier certificate $B(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ satisfies Eq. (6).

Proof. Proposition 1 and Theorem 1 entail soundness and completeness, respectively. □

In fact, in most practical cases, Eq. (6) with $\epsilon = 0$ can be viewed as a *sound and complete* characterization. In this situation, completeness follows from the so-called "finite convergence property" of Theorem 1, which requires the underlying basic semialgebraic sets $\mathcal{X}$, $\mathcal{I}$, and $\mathcal{U}$ to satisfy some side conditions that are generally true [26]. For now, we do not go deep into these details and just consider that soundness and completeness are dependent on the parameter ϵ.

Unfortunately, when the domain $\mathcal{X}$ becomes unbounded, Eq. (6) with $\epsilon > 0$ is no longer a complete characterization due to the violation of the Archimedean condition, while the $\epsilon = 0$ case is still sound. Consequently, we can solely rely on the sound characterization to synthesize barrier certificates, which may fail to identify potential solutions as in the following example. So the problem considered in this paper is, **can we derive a complete characterization similar to Eq. (6) for the unbounded cases?**

Example 2. Consider an 1-dimensional system $f(x_1) = x_1$ with $\mathcal{X} = \mathbb{R}$, $\mathcal{I} = \{x_1 \mid x_1^3 \geq 0\}$, and $\mathcal{U} = \{x_1 \mid x_1 + 1 \leq 0\}$, then $B(x_1) = -x_1$ is a barrier certificate but is not a solution to Eq. (6) with $\epsilon = 0$. To see this, we only need to show that there exists no sum-of-squares polynomials $\sigma_0^{\mathcal{I}}(x_1), \sigma_1^{\mathcal{I}}(x_1) \in \Sigma[x_1]$ such that $x_1 = \sigma_0^{\mathcal{I}}(x_1) + x_1^3\sigma_1^{\mathcal{I}}(x_1)$. Suppose we have such an expression, by setting $x_1 = 0$, we have $\sigma_0^{\mathcal{I}}(0) = 0$. Assume that $\sigma_0^{\mathcal{I}}$ can be expressed as $\sigma_0^{\mathcal{I}}(x_1) = \sum_i p_i^2(x_1)$, then $\sigma_0^{\mathcal{I}}(0) = 0$ implies that $p_i(0) = 0$ for each i, so each p_i factors as $p_i(x_1) = x_1 p_i'(x_1)$. Therefore, both $\sigma_0^{\mathcal{I}}(x_1)$ and $x_1^3\sigma_1^{\mathcal{I}}(x_1)$ contain no terms of degree less than 2, which is impossible.

4 A Complete Characterization of Polynomial Barrier Certificates

In this section, we give an affirmative answer to the question raised above. The tool we use is a newly introduced technique in the optimization community, called *homogenization* [19], to transform an unbounded optimization problem into a bounded one. In the following, we utilize the homogenization technique to derive a complete characterization for polynomial barrier certificates purely from a constraint-solving perspective. In the next section, we will take a different view of this technique and consider a family of non-polynomial barrier certificates that arise naturally.

We first fix some notations. Given $\boldsymbol{x} \in \mathbb{R}^n$, let x_0 be a fresh variable. For a polynomial $p(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ of degree d, its homogenization w.r.t. variable x_0 is a new polynomial $\tilde{p} \in \mathbb{R}[x_0, \boldsymbol{x}]$ defined by $\tilde{p}(x_0, \boldsymbol{x}) \;\hat{=}\; x_0^d p(x_1/x_0, \ldots, x_n/x_0)$. For example, let $f(x_1, x_2) = x_1^2 + x_2 + 1$, then $\tilde{f}(x_0, x_1, x_2) = x_1^2 + x_2 x_0 + x_0^2$. Suppose $\mathcal{K} \subseteq \mathbb{R}^n$ is a semialgebraic set as described in Eq. (1), we introduce two related sets in $\mathbb{R}^{n+1}$ as follows:

$$\begin{aligned} \tilde{\mathcal{K}}_{>0} \;&\hat{=}\; \left\{(x_0, \boldsymbol{x}) \mid \tilde{p}_1(x_0, \boldsymbol{x}) \geq 0, \ldots, \tilde{p}_m(x_0, \boldsymbol{x}) \geq 0, \|\boldsymbol{x}\|^2 + x_0^2 = 1, x_0 > 0\right\}, \\ \tilde{\mathcal{K}} \;&\hat{=}\; \left\{(x_0, \boldsymbol{x}) \mid \tilde{p}_1(x_0, \boldsymbol{x}) \geq 0, \ldots, \tilde{p}_m(x_0, \boldsymbol{x}) \geq 0, \|\boldsymbol{x}\|^2 + x_0^2 = 1, x_0 \geq 0\right\}. \end{aligned}$$

One can see that there exists an one-to-one mapping between $\tilde{\mathcal{K}}_{>0}$ and $\mathcal{K}$:

Lemma 1. *Let $\mathcal{K}$ be as in Eq. (1). Then $\boldsymbol{x} \in \mathcal{K}$ if and only if*

$$\left(\frac{1}{\sqrt{1 + \|\boldsymbol{x}\|^2}}, \frac{x_1}{\sqrt{1 + \|\boldsymbol{x}\|^2}}, \ldots, \frac{x_n}{\sqrt{1 + \|\boldsymbol{x}\|^2}}\right) \in \tilde{\mathcal{K}}_{>0}.$$

Moreover, $(x_0, \boldsymbol{x}) \in \tilde{\mathcal{K}}_{>0}$ if and only if $(\frac{x_1}{\sqrt{1-\|\boldsymbol{x}\|^2}}, \ldots, \frac{x_n}{\sqrt{1-\|\boldsymbol{x}\|^2}}) \in \mathcal{K}$.

Utilizing the above lemma, we can transform a potentially *unbounded* set into a *bounded* set located on the unit sphere within $\mathbb{R}^{n+1}$. Moreover, note that points with $x_0 = 0$ in $\mathbb{R}^{n+1}$ correspond to points at infinity in $\mathbb{R}^n$. This encourages us to take the points at infinity into consideration. The related concept is captured by the following definition.

Definition 1 (Closed at Infinity [26]**).** *A basic semialgebraic set $\mathcal{K}$ is closed at infinity if $cl(\tilde{\mathcal{K}}_{>0}) = \tilde{\mathcal{K}}$, where $cl(\tilde{\mathcal{K}}_{>0})$ denotes the closure of $\tilde{\mathcal{K}}_{>0}$.*

We would like to emphasize that being closed at infinity is a generic property for semialgebraic sets [15], and its manifestation may be contingent upon the selection of descriptive polynomials. For example, let $S_1 = \{(x_1, x_2) \mid x_1 - x_2^2 \geq 0\}$, then S_1 is not closed at infinity because

$$(0, -1, 0) \notin cl(\tilde{S}_{1>0}) \quad \text{and} \quad (0, -1, 0) \in \tilde{S}_1.$$

However, by adding a redundant polynomial inequality $x_1 \geq 0$ in S_1, we can check $S_2 = \{(x_1, x_2) \in \mathbb{R}^2 \mid x_1 - x_2^2 \geq 0, x_1 \geq 0\}$ $(= S_1)$ is closed at infinity. In this paper, we assume that $\mathcal{I}, \mathcal{U}$, and $\mathcal{X}$ are all closed at infinity, which is purely a technical assumption. To check whether a semialgebraic set is closed at ∞, one can use [15, Thm. 2.11].

The following theorem lies at the core of the homogenization technique.

Theorem 4 ([19, Lem 3.2]**).** *When a basic semialgebraic set $\mathcal{K}$ is closed at infinity, for any polynomial $f \in \mathbb{R}[\boldsymbol{x}]$*

$$f(\boldsymbol{x}) \geq 0 \text{ over } \mathcal{K} \iff \tilde{f}(x_0, \boldsymbol{x}) \geq 0 \text{ over } \tilde{\mathcal{K}}.$$

Now we present the homogenized version of Theorem 3, which solves the problem raised at the end of the last section.

Theorem 5. *Assume that $\mathcal{I}$, $\mathcal{U}$, and $\mathcal{X}$ are all closed at infinity. Given $\lambda \in \mathbb{R}$ and $\epsilon_e \in \mathbb{R}_{>0}$, consider the following constraints with parameter ϵ,*

$$\begin{aligned}
&-\tilde{B}(x_0, \boldsymbol{x}) + \epsilon = \sigma_0^{\mathcal{I}} + \sum_{i=1}^{m_i+2} \sigma_i^{\mathcal{I}} \tilde{g}_i^{\mathcal{I}} \\
&\tilde{B}(x_0, \boldsymbol{x}) - \epsilon_e x_0^d + \epsilon = \sigma_0^{\mathcal{U}} + \sum_{i=1}^{m_u+2} \sigma_i^{\mathcal{U}} \tilde{g}_i^{\mathcal{U}} \\
&\tilde{H}(x_0, \boldsymbol{x}) + \epsilon = \sigma_0^{\mathcal{X}} + \sum_{i=1}^{m_x+2} \sigma_i^{\mathcal{X}} \tilde{g}_i^{\mathcal{X}} \\
&\sigma_0^{\mathcal{I}}, \ldots, \sigma_{m_i+1}^{\mathcal{I}}, \sigma_0^{\mathcal{U}}, \ldots, \sigma_{m_u+1}^{\mathcal{U}}, \sigma_0^{\mathcal{X}}, \ldots, \sigma_{m_x+1}^{\mathcal{X}} \in \Sigma[x_0, \boldsymbol{x}], \\
&\sigma_{m_i+2}^{\mathcal{I}}, \sigma_{m_u+2}^{\mathcal{U}}, \sigma_{m_x+2}^{\mathcal{X}} \in \mathbb{R}[x_0, \boldsymbol{x}],
\end{aligned} \tag{7}$$

where $H(\boldsymbol{x}) \hat{=} \lambda B(\boldsymbol{x}) - \mathfrak{L}_{\boldsymbol{f}} B(\boldsymbol{x})$, d is the degree of $\deg B(\boldsymbol{x})$*, $\tilde{g}_{m_i+1}^{\mathcal{I}} = \tilde{g}_{m_u+1}^{\mathcal{U}} = \tilde{g}_{m_x+1}^{\mathcal{X}} = x_0$, and $\tilde{g}_{m_i+2}^{\mathcal{I}} = \tilde{g}_{m_u+2}^{\mathcal{U}} = \tilde{g}_{m_x+2}^{\mathcal{X}} = x_0^2 + \|\boldsymbol{x}\|^2 - 1$. When $\epsilon = 0$, Eq. (7)*

gives a sound characterization of polynomial barrier certificates, i.e., any solution $B(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ of degree d to the above constraints is a barrier certificate. When $\epsilon > 0$, Eq. (7) gives a complete characterization of polynomial barrier certificates, i.e., any barrier certificate $B(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ of degree d satisfies the above constraints.

Proof. We prove the first constraint corresponding to the initial set $\mathcal{I}$, the other two constraints are similar. By employing homogenization and Theorem 4, the original condition Eq. (3) can be transformed into $-\tilde{B}(x_0, \boldsymbol{x}) \geq 0$ over $\tilde{\mathcal{I}}$. Since the descriptive polynomials in $\tilde{\mathcal{I}}$ contain $\|\boldsymbol{x}\|^2 + x_0^2 = 1$, $\tilde{\mathcal{I}}$ is a closed basic semi-algebraic set and its corresponding quadratic module is Archimedean. Thus, we can apply Proposition 1 and Theorem 1 to obtain the soundness and completeness results, respectively. Note for the other two constraints, we need to homogenize the polynomial $B(\boldsymbol{x}) - \epsilon_e$ and $B(\boldsymbol{x}) - \mathfrak{L}_{\boldsymbol{f}} B(\boldsymbol{x})$ as a whole. □

5 Homogenized Systems and Semialgebraic Barrier Certificates

In this section, we take a different perspective of the technique in the last section. The motivation comes from the observation that the homogenization procedure can be viewed as mapping the original system in $\mathbb{R}^n$ into a new system in $\mathbb{R}^{n+1}$. Consequently, the constraints in Eq. (7) can be conceived as barrier certificate conditions for the new system. Employing this idea, we introduce the definition of homogenized systems as follows. To avoid confusion, we will use $(y_0, \boldsymbol{y}) \in \mathbb{R}^{n+1}$ to denote the state variables of the homogenized systems.

Definition 2 (Homogenized System). *Given a system Eq. (2), the homogenized system is an associated system in $\mathbb{R}^{n+1}$. For each state $\boldsymbol{x} \in \mathbb{R}^n$ of the original system, the corresponding state $(y_0, \boldsymbol{y}) \in \mathbb{R}^{n+1}$ of the homogenized system is given by*

$$(y_0, y_1, \ldots, y_n) = (\frac{1}{\sqrt{1 + \|\boldsymbol{x}\|^2}}, \frac{x_1}{\sqrt{1 + \|\boldsymbol{x}\|^2}}, \ldots, \frac{x_n}{\sqrt{1 + \|\boldsymbol{x}\|^2}}). \tag{8}$$

The dynamics of the homogenized systems can be obtained by taking derivative in the right-hand-side of Eq. (8). Hence, the safety verification problem of the original system Eq. (2) with sets $\mathcal{X}$, $\mathcal{I}$, and $\mathcal{U}$ can be translated into an equivalent problem for the homogenized system Eq. (8) with sets $\tilde{\mathcal{X}}$, $\tilde{\mathcal{I}}$, and $\tilde{\mathcal{U}}$. Furthermore, we show that a barrier certificate of the original system can be computed from a barrier certificate of the homogenized system.

Theorem 6. *$B(y_0, \boldsymbol{y}) \in \mathcal{C}^1(\mathbb{R}^{n+1})$ is a barrier certificate of the homogenized system if and only if $B(\frac{1}{\sqrt{\|\boldsymbol{x}\|^2+1}}, \frac{\boldsymbol{x}}{\sqrt{\|\boldsymbol{x}\|^2+1}})$ is a barrier certificate of the original system.*

Proof. Let $B(y_0, \boldsymbol{y})$ be a barrier certificate of the homogenized system. Denote $g(\boldsymbol{x}) \hat{=} B(\frac{1}{\sqrt{\|\boldsymbol{x}\|^2+1}}, \frac{\boldsymbol{x}}{\sqrt{\|\boldsymbol{x}\|^2+1}})$, we show that $g(\boldsymbol{x})$ satisfies the conditions in

Theorem 2. For $\boldsymbol{x} \in \mathcal{I}$, since $(y_0, \boldsymbol{y}) \in \tilde{\mathcal{I}}^b$ by Lem. 1 and Eq. (8), we have $g(\boldsymbol{x}) = B(y_0, \boldsymbol{y}) \leq 0$. Similarly, for $\boldsymbol{x} \in \mathcal{U}$, we have $g(\boldsymbol{x}) = B(y_0, \boldsymbol{y}) \geq \epsilon_e$. Finally, since

$$\begin{aligned}\mathfrak{L}_{\boldsymbol{f}} g(\boldsymbol{x}) &= \sum_{i=1}^{n} \frac{\partial g(\boldsymbol{x})}{\partial x_i} f_i(\boldsymbol{x}) = \sum_{i=1}^{n} \left(\sum_{j=0}^{n} \frac{\partial B(y_0, \boldsymbol{y})}{\partial y_j} \frac{\partial y_j}{\partial x_i} \right) f_i(\boldsymbol{x}) \\ &= \sum_{j=0}^{n} \frac{\partial B(y_0, \boldsymbol{y})}{\partial y_j} \left(\sum_{i=1}^{n} \frac{\partial y_j}{\partial x_i} f_i(\boldsymbol{x}) \right) = \mathfrak{L}_{\boldsymbol{f}'} B(y_0, \boldsymbol{y}),\end{aligned}$$

where $\boldsymbol{f}'$ is the dynamic of the homogenized system. For any $\boldsymbol{x} \in \mathcal{X}$ we have $\mathfrak{L}_{\boldsymbol{f}} g(\boldsymbol{x}) - \lambda g(\boldsymbol{x}) = \mathfrak{L}_{\boldsymbol{f}'} B(y_0, \boldsymbol{y}) - \lambda B(y_0, \boldsymbol{y}) \leq 0$. The other direction is similar. $\square$

According to Stone-Weierstrass theorem [38], a continuous function in a compact space in $\mathbb{R}^{n+1}$ can be approximated by polynomials. This means that, if there exists $B(y_0, \boldsymbol{y}) \in \mathcal{C}^1(\mathbb{R}^{n+1})$ as a barrier certificate, one should be able to find a polynomial barrier certificate (of sufficient large degree) close to it. In fact, this is one of the reasons why we are primarily concerned with polynomial barrier certificates in the bounded case. By Theorem 6, if $B(y_0, \boldsymbol{y})$ is a polynomial of degree d, then we have

$$(\sqrt{\|\boldsymbol{x}\|^2 + 1})^d B(\frac{1}{\sqrt{\|\boldsymbol{x}\|^2 + 1}}, \frac{\boldsymbol{x}}{\sqrt{\|\boldsymbol{x}\|^2 + 1}}) = B_1(\boldsymbol{x}) + \sqrt{\|\boldsymbol{x}\|^2 + 1} \cdot B_2(\boldsymbol{x})$$

for some polynomials $B_1(\boldsymbol{x}), B_2(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$. From this expression, we can see that Theorem 5 is a special case when $B(y_0, \boldsymbol{y})$ itself is a homogeneous polynomial (i.e., all monomials are of the same degree) and $B_2(\boldsymbol{x}) = 0$.

Definition 3. *We say a barrier certificate $B(\boldsymbol{x})$ is* semialgebraic [1] *if it can be expressed as $B(\boldsymbol{x}) = B_1(\boldsymbol{x}) + \sqrt{\|\boldsymbol{x}\|^2 + 1} \cdot B_2(\boldsymbol{x})$ for some $B_1(\boldsymbol{x}), B_2(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$.*

The synthesis of semialgebraic barrier certificates is not straightforward, due to the existence of non-polynomial component $\sqrt{\|\boldsymbol{x}\|^2 + 1}$. To address this problem, we employ the technique in [22] to encode these non-polynomial expressions into polynomials with extra variables. To be concrete, we introduce two variables u and v, which stand for $\sqrt{\|\boldsymbol{x}\|^2 + 1}$ and $\frac{1}{\sqrt{\|\boldsymbol{x}\|^2 + 1}}$, respectively. Then, by Theorem 2, the conditions for a semialgebraic barrier certificate can be written as

$$\begin{aligned} B_1(\boldsymbol{x}) + uB_2(\boldsymbol{x}) &\leq 0, &&\text{for } \boldsymbol{x} \in \mathcal{I}, u^2 = \|\boldsymbol{x}\|^2 + 1, u \geq 0, \\ B_1(\boldsymbol{x}) + uB_2(\boldsymbol{x}) &\geq \epsilon_e, &&\text{for } \boldsymbol{x} \in \mathcal{U}, u^2 = \|\boldsymbol{x}\|^2 + 1, u \geq 0, \\ G(\boldsymbol{x}, u, v) &\geq 0, &&\text{for } \boldsymbol{x} \in \mathcal{X}, u^2 = \|\boldsymbol{x}\|^2 + 1, u \geq 0, uv = 1, \end{aligned} \tag{9}$$

[1] A function $f(\boldsymbol{x})$ is called semialgebraic if its graph $\{(\boldsymbol{x}, f(\boldsymbol{x})) \mid \boldsymbol{x} \in \mathbb{R}^n\}$ is a semialgebraic set. Here, semialgebraic barrier certificates can only express a certain subclass of semialgebraic functions.

where $G(\boldsymbol{x}, u, v) \in \mathbb{R}[\boldsymbol{x}, u, v]$ is defined by

$$\begin{aligned}
&\lambda \left(B_1(\boldsymbol{x}) + \sqrt{\|\boldsymbol{x}\|^2+1} \cdot B_2(\boldsymbol{x})\right) - \mathfrak{L}_{\boldsymbol{f}} \left(B_1(\boldsymbol{x}) + \sqrt{\|\boldsymbol{x}\|^2+1} \cdot B_2(\boldsymbol{x})\right) \\
= \;& \lambda \left(B_1(\boldsymbol{x}) + u \cdot B_2(\boldsymbol{x})\right) - \mathfrak{L}_{\boldsymbol{f}} B_1(\boldsymbol{x}) - u \cdot \mathfrak{L}_{\boldsymbol{f}} B_2(\boldsymbol{x}) - v B_2(\boldsymbol{x}) \sum_{i=1}^{n} x_i f_i(\boldsymbol{x}) \\
\hat{=} \;& G(\boldsymbol{x}, u, v).
\end{aligned} \tag{10}$$

Similar to Theorems 3 and 5, we have the following characterization for semialgebraic barrier certificates. Without loss of generality, we assume that $B_1(\boldsymbol{x})$ and $B_2(\boldsymbol{x})$ are both of degree d.

Theorem 7. *Assume that $\mathcal{I}$, $\mathcal{U}$, and $\mathcal{X}$ are all closed at infinity. Given $\lambda \in \mathbb{R}$ and $\epsilon_e \in \mathbb{R}_{>0}$, consider the following constraints with parameter ϵ,*

$$\begin{aligned}
& B(\boldsymbol{x}, u) = B_1(\boldsymbol{x}) + u \cdot B_2(\boldsymbol{x}) \\
& -\tilde{B}(x_0, \boldsymbol{x}, u) + \epsilon = \sigma_0^{\mathcal{I}} + \sum_{i=1}^{m_i+4} \sigma_i^{\mathcal{I}} \tilde{g}_i^{\mathcal{I}} \\
& \tilde{B}(x_0, \boldsymbol{x}, u) - \epsilon_e x_0^{d+1} + \epsilon = \sigma_0^{\mathcal{U}} + \sum_{i=1}^{m_u+4} \sigma_i^{\mathcal{U}} \tilde{g}_i^{\mathcal{U}} \\
& \tilde{G}(x_0, \boldsymbol{x}, u, v) + \epsilon = \sigma_0^{\mathcal{X}} + \sum_{i=1}^{m_x+5} \sigma_i^{\mathcal{X}} \tilde{g}_i^{\mathcal{X}} \\
& \sigma_0^{\mathcal{I}}, \ldots, \sigma_{m_i+2}^{\mathcal{I}}, \sigma_0^{\mathcal{U}}, \ldots, \sigma_{m_u+2}^{\mathcal{U}} \in \Sigma[x_0, \boldsymbol{x}, u], \\
& \sigma_0^{\mathcal{X}}, \ldots, \sigma_{m_x+2}^{\mathcal{X}} \in \Sigma[x_0, \boldsymbol{x}, u, v], \\
& \sigma_{m_i+3}^{\mathcal{I}}, \sigma_{m_i+4}^{\mathcal{I}}, \sigma_{m_u+3}^{\mathcal{U}}, \sigma_{m_u+4}^{\mathcal{U}} \in \mathbb{R}[x_0, \boldsymbol{x}, u] \\
& \sigma_{m_x+3}^{\mathcal{X}}, \sigma_{m_x+4}^{\mathcal{X}}, \sigma_{m_x+5}^{\mathcal{X}} \in \mathbb{R}[x_0, \boldsymbol{x}, u, v],
\end{aligned} \tag{11}$$

where $G(\boldsymbol{x}, u, v)$ is as defined in Eq. (10), $\tilde{g}_{m_i+1}^{\mathcal{I}} = \tilde{g}_{m_u+1}^{\mathcal{U}} = \tilde{g}_{m_x+1}^{\mathcal{X}} = x_0$, $\tilde{g}_{m_i+2}^{\mathcal{I}} = \tilde{g}_{m_u+2}^{\mathcal{U}} = \tilde{g}_{m_x+2}^{\mathcal{X}} = u$, $\tilde{g}_{m_i+3}^{\mathcal{I}} = \tilde{g}_{m_u+3}^{\mathcal{U}} = \tilde{g}_{m_x+3}^{\mathcal{X}} = u^2 - x_0^2 - \|\boldsymbol{x}\|^2$, $\tilde{g}_{m_i+4}^{\mathcal{I}} = \tilde{g}_{m_u+4}^{\mathcal{U}} = x_0^2 + \|\boldsymbol{x}\|^2 + u^2 - 1$, $\tilde{g}_{m_x+4}^{\mathcal{X}} = uv - x_0^2$, and $\tilde{g}_{m_x+5}^{\mathcal{X}} = x_0^2 + \|\boldsymbol{x}\|^2 + u^2 + v^2 - 1$. When $\epsilon = 0$, Eq. (11) gives a sound characterization for semialgebraic barrier certificates, i.e., any pair of solutions $B_1(\boldsymbol{x}), B_2(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ to the above constraints makes $B(\boldsymbol{x})$ a barrier certificate. When $\epsilon > 0$, Eq. (11) gives a complete characterization for semialgebraic barrier certificates, i.e., any semialgebraic barrier certificate with $B_1(\boldsymbol{x}), B_2(\boldsymbol{x}) \in \mathbb{R}[\boldsymbol{x}]$ of degree d satisfies the above constraints.

Proof. By applying Theorem 4 to Eq. (9). Similar to the proof of Theorem 5. □

6 Experiments

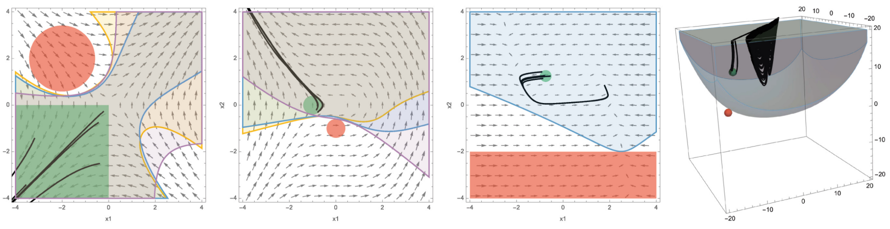
Implementation. We implemented the barrier certificate synthesis procedures in JULIA programming language, interfaced with TSSOS [40] for formulating SOS relaxations and MOSEK solver [5] for solving the underlying SDP. All experiments were performed on a Mac lap-top with Apple M2 chip and 8GB memory. The code and benchmarks are publicly available online . In the following, we use the corresponding theorems to refer to different approaches/characterizations.

Experiment Settings. The goal of our experiments was to compare the differences between employing characterizations Theorems 3, 5, and 7 to synthesize exponential-type barrier certificates over unbounded domains. To this end, we collected a set of dynamical systems of dimension 2 and 3 from the literature. For each benchmark system, we designed two problem instances. In the first instance, we only let the domain $\mathcal{X} = \mathbb{R}^n$ be unbounded, while in the second instance, we further let the initial set $\mathcal{I}$ and/or the unsafe region $\mathcal{U}$ be unbounded (not necessarily containing the original bounded counterparts).

In practical computation, we set $\lambda = -1$, $\epsilon_e = 10^{-5}$ in the definition of barrier certificates and $\epsilon = 0$ in the sum-of-squares characterizations. As discussed after Theorem 3, the $\epsilon = 0$ case can be viewed as both sound and complete in most practical situations. We manually verified that the sets $\mathcal{I}$, $\mathcal{U}$, and $\mathcal{X}$ are closed at infinity.

For Theorems 3 and 5, we searched for polynomial barrier certificates $B(\boldsymbol{x})$ up to degree 6. For Theorem 7, due to the $\sqrt{\|\boldsymbol{x}\|^2 + 1}$ term, we searched for semi-algebraic barrier certificates with $B_1(\boldsymbol{x})$ and $B_2(\boldsymbol{x})$ up to degree 4. When the target degree d is fixed, by restricting the highest degree of involved polynomials to be the smallest even number larger than d, the sum-of-squares characterizations can be solved as SDPs [7]. For each solution returned by SDP solver, we utilized MATHEMATICA to symbolically verify that the numerical solution $B(\boldsymbol{x})$ satisfies the barrier certificate conditions. The timeout for verifying each barrier certificate candidate was set to be 10 min. We report the total time for solving SDP constraints and verifying the results.

Empirical Observations. Table 1 reports the experimental results, and Fig. 1 portraits selected examples. We mainly compare the results from two aspects.



(a) vector-1 (b) lie-der-1 (c) nagumo-2 (d) lorenz-1

green region: initial set $\mathcal{I}$; red region: unsafe region $\mathcal{U}$; black solid curves: sampled trajectories $\xi_{\boldsymbol{x}_0}$; light blue, yellow, and purple region: sub-level set of the synthesized barrier certificate by using Thm. 3, Thm. 5, and Thm. 7.

Fig. 1. Portraits of four selected examples.

Table 1. Experimental results for synthesizing exponential type barrier certificates.

			Thm. 3 [21]			Our Thm. 5			Our Thm. 7		
system	dim	unbounded	deg	succ	time(s)	deg	succ	time(s)	deg	succ	time(s)
vector[37]	2	$\mathcal{X}$	4	✓	0.58	3	✓	0.03	**2**	✓	0.83
		$\mathcal{I},\mathcal{U},\mathcal{X}$	> 6	✗	0.39	4	✓	0.16	**2**	✓	0.72
barrier[29]	2	$\mathcal{X}$	> 6	✗	2.40	> 6	✗	2.73	>4	✗	22.98
		$\mathcal{I},\mathcal{U},\mathcal{X}$	> 6	✗	0.78	3	✓	0.04	**2**	✓	3.12
lie-der[24]	2	$\mathcal{X}$	3	✓	0.19	3	✓	0.14	**1**	✓	0.75
		$\mathcal{I},\mathcal{U},\mathcal{X}$	**3**	✓	0.14	**3**	✓	0.19	**3**	✓	5.04
arch1[36]	2	$\mathcal{X}$	**4**	✓	0.40	**4**	✓	0.54	> 4	✗	117.61
		$\mathcal{I},\mathcal{U},\mathcal{X}$	**1**	✓	0.09	**1**	✓	0.04	2	✓	20.92
arch2[36]	2	$\mathcal{X}$	**3**	✓	0.20	**3**	✓	0.21	**3**	✓	5.53
		$\mathcal{I},\mathcal{U},\mathcal{X}$	3	✓	0.18	3	✓	0.18	**2**	✓	1.59
arch3[36]	2	$\mathcal{X}$	**2**	✓	0.12	**2**	✓	0.13	**2**	✓	3.45
		$\mathcal{I},\mathcal{U},\mathcal{X}$	> 6	✗	0.84	> 6	✗	1.30	**1**	✓	0.34
arch4[36]	2	$\mathcal{X}$	> 6	✗	0.11	5	✓	0.74	**3**	✓	5.69
		$\mathcal{U},\mathcal{X}$	> 6	✗	1.02	**6**	✓	1.21	> 4	✗	7.74
nagumo[34]	2	$\mathcal{X}$	**2**	✓	0.13	**2**	✓	0.14	**2**	✓	3.50
		$\mathcal{U},\mathcal{X}$	> 6	✗	1.02	**3**	✓	0.17	> 4	✗	23.73
lorenz[10]	3	$\mathcal{X}$	6	?	TO	**4**	✓	72.13	2	?	TO
		$\mathcal{U},\mathcal{X}$	**5**	✓	248.88	6	?	TO	2	?	TO
lotka[14]	3	$\mathcal{X}$	> 6	✗	88.97	> 6	✗	27.8	3	?	TO
		$\mathcal{U},\mathcal{X}$	> 6	✗	0.27	> 6	✗	438.54	3	?	TO

dim: system dimension; **unbounded**: the unbounded region(s); **deg**: degree of polynomial barrier certificates or polynomial components in semialgebraic barrier certificates; **succ**: whether our algorithm succeeds in finding a valid barrier certificate. ✓ means valid solution, ✗ means no solution or invalid solution (within the search range), and ? means unverified; TO: verification takes more than 10 min in MATHEMATICA; **time**: total time for SDP solving and verification.

Expressiveness : For problems with unbounded domains, both our complete characterizations Theorems 5 and 7 are more expressive than the incomplete characterization Theorem 3, as they succeeds in synthesizing barrier certificates in more problem instances. The two complete characterizations offer distinct advantages: Thm. 5 exhibits broader applicability, demonstrably successful for problem instances like arch4-2 and nagumo-2. In contrast, Theorem 7 excels at synthesizing lower-degree barrier certificates, as exemplified by vector-1,2 and barrier-1,2 problem instances. The experimental results also demonstrate that, while Theorem 7 theoretically subsumes Theorem 5, its characterization presents significantly greater complexity and hinders its ability to identify solutions, due to inherent numerical issues in SDP solvers.

Efficiency : For most benchmarks, the time overhead for employing Theorem 5 is comparable to Theorem 3, while Theorem 7 is evidently slower than the other two. This should be attributed to the introduction of fresh variables in SOS characterizations in the complete characterizations (one for Theorem 5 and two for Theorem 7). Hence, the computation cost of both SDP solving and posterior verification increases, mildly for Theorem 5 (e.g., lie-der-2 and arch1-1) but severely for Theorem 7 (e.g., barrier-1 and arch1-1). We also want to emphasize that, for 3-dimensional systems with higher-degree templates, posterior verification time increases significantly, meaning that we can not verify the validity of the barrier certificate candidates within a reasonable amount of time.

Summary. For practical applications, we recommend employing Theorem 5 to synthesize polynomial barrier certificates for unbounded problems. This approach achieves a high level of expressiveness while maintaining efficiency comparable to Theorem 3. Moreover, we believe that the performance of Theorem 7 can be improved by exploiting algebraic structures of the constraints. For example, the variables u, v only occur linearly or quadratically in constraints, which can be utilized in restrict the templates of unknown sum-of-squares polynomials.

Remark 1. In our experiments, we did *not* consider different parameter settings (such as the selection of λ discussed in [21]) and constraint formulations (such as techniques for taming numerical errors discussed in [32]), which may impact the synthesized barrier certificates but are not the focus of the current paper.

7 Conclusion

This paper addresses the problem of synthesizing barrier certificates over unbounded domains. Previous SDP-based approaches to this problem are incomplete, because Putinar's Positivstellensatz is only applicable in bounded cases. We fill this gap by proposing the first complete sum-of-squares characterization for polynomial barrier certificates, achieved through the utilization of the homogenization approach derived from optimization theory. Furthermore, we introduce the notions of homogenized systems and semialgebraic barrier certificates, which are induced from polynomial barrier certificates of the homogenized systems. For such non-polynomial barrier certificates, we also provide a complete characterization. Experimental results substantiate the efficacy of both of our approaches, demonstrating their enhanced expressiveness and ability to synthesize more barrier certificates in comparison to existing methods.

While our paper primarily focuses on synthesizing barrier certificates for differential dynamical systems, it is crucial to note that our method can be readily extended to other types of systems, including hybrid systems and systems with control, disturbance, or stochastic dynamics. Furthermore, our method can also be utilized in related verification problems such as Lyapunov function synthesis, program invariant generation, and so on.

Acknowledgments. This work has been partially funded by the National Key R&D Program of China under grant No. 2022YFA1005101 and 2022YFA1005102, by the NSFC under grant No. 62192732 and 62032024, by the CAS Project for Young Scientists in Basic Research under grant No. YSBR-040, by the Strategic Priority Research Program of the Chinese Academy of Sciences XDB0640000 & XDB0640200, by the Key R&D Program of Hubei Province (2023BAB170), and by the Fundamental Research Funds for the Central Universities.

Data Availability Statement. The experimental results of this paper may be reproduced using the artifact on Figshare https://doi.org/10.6084/m9.figshare.26085853, or via GitHub link https://github.com/EcstasyH/BCunbounded.

References

1. Abate, A., Ahmed, D., Edwards, A., Giacobbe, M., Peruffo, A.: FOSSIL: a software tool for the formal synthesis of Lyapunov functions and barrier certificates using neural networks. In: 24th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2021, pp. 1–11. ACM (2021)
2. Ames, A.D., Xu, X., Grizzle, J.W., Tabuada, P.: Control barrier function based quadratic programs for safety critical systems. IEEE Trans. Autom. Control **62**(8), 3861–3876 (2017)
3. Anand, M., Murali, V., Trivedi, A., Zamani, M.: Safety verification of dynamical systems via k-inductive barrier certificates. In: 2021 60th IEEE Conference on Decision and Control CDC 2021, pp. 1314–1320. IEEE (2021)
4. Anand, M., Murali, V., Trivedi, A., Zamani, M.: k-inductive barrier certificates for stochastic systems. In: Hybrid Systems: Computation and Control, 25th ACM International Conference, pp. 1–11. ACM (2022)
5. ApS, M.: MOSEK Optimizer API for Julia. Version 10.1.13. (2019). https://docs.mosek.com/latest/juliaapi/index.html
6. Bak, S.: t-barrier certificates: a continuous analogy to k-induction. In: 6th IFAC Conference on Analysis and Design of Hybrid Systems, ADHS 2018. IFAC-PapersOnLine, vol. 51, pp. 145–150. Elsevier (2018)
7. Blekherman, G., Parrilo, P.A., Thomas, R.R.: Semidefinite Optimization and Convex Algebraic Geometry. SIAM (2012)
8. Dai, L., Gan, T., Xia, B., Zhan, N.: Barrier certificates revisited. J. Symb. Comput. **80**, 62–86 (2017)
9. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. J. Symb. Comput. **5**(1–2), 29–35 (1988)
10. Djaballah, A., Chapoutot, A., Kieffer, M., Bouissou, O.: Construction of parametric barrier functions for dynamical systems using interval analysis. Automatica **78**, 287–296 (2017)
11. Feng, S., Chen, M., Xue, B., Sankaranarayanan, S., Zhan, N.: Unbounded-time safety verification of stochastic differential dynamics. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 327–348. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_18
12. Gao, S., Avigad, J., Clarke, E.M.: δ-complete decision procedures for satisfiability over the reals. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 286–300. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_23

13. Gao, S., Kong, S., Clarke, E.M.: dReal: an smt solver for nonlinear theories over the reals. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 208–214. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_14
14. Goubault, E., Jourdan, J., Putot, S., Sankaranarayanan, S.: Finding non-polynomial positive invariants and lyapunov functions for polynomial systems through darboux polynomials. In: American Control Conference, ACC 2014, pp. 3571–3578. IEEE (2014)
15. Guo, F., Wang, L., Zhou, G.: Minimizing rational functions by exact jacobian SDP relaxation applicable to finite singularities. J. Global Optim. **58**(2), 261–284 (2014)
16. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, STOC 1995, pp. 373–382. ACM (1995)
17. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
18. Huang, C., Chen, X., Lin, W., Yang, Z., Li, X.: Probabilistic safety verification of stochastic hybrid systems using barrier certificates. ACM Trans. Embed. Comput. Syst. **16**(5s), 1–19 (2017)
19. Huang, L., Nie, J., Yuan, Y.: Homogenization for polynomial optimization with unbounded sets. Math. Program. **200**(1), 105–145 (2023)
20. Jagtap, P., Soudjani, S., Zamani, M.: Formal synthesis of stochastic systems via control barrier certificates. IEEE Trans. Autom. Control **66**(7), 3097–3110 (2021)
21. Kong, H., He, F., Song, X., Hung, W.N.N., Gu, M.: Exponential-condition-based barrier certificate generation for safety verification of hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 242–257. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_17
22. Lasserre, J.B., Putinar, M.: Positivity and optimization: beyond polynomials. In: Anjos, M.F., Lasserre, J.B. (eds.) Handbook on Semidefinite, Conic and Polynomial Optimization. International Series in Operations Research & Management Science, vol. 166, pp. 407–434. Springer, New York, NY (2012). https://doi.org/10.1007/978-1-4614-0769-0_14
23. Lasserre, J.B.: Moments, Positive Polynomials and their Applications, vol. 1. World Scientific (2009)
24. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. In: Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, pp. 97–106. ACM (2011)
25. Murali, V., Trivedi, A., Zamani, M.: Closure certificates. In: HSCC 2024: Proceedings of the 27th ACM International Conference on Hybrid Systems: Computation and Control, pp. 1–11 (2024)
26. Nie, J.: Discriminants and nonnegative polynomials. J. Symb. Comput. **47**(2), 167–191 (2012)
27. Peruffo, A., Ahmed, D., Abate, A.: Automated and formal synthesis of neural barrier certificates for dynamical models. In: TACAS 2021. LNCS, vol. 12651, pp. 370–388. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_20
28. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 176–189. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_17
29. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_32

30. Prajna, S., Jadbabaie, A., Pappas, G.J.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE Trans. Autom. Control **52**(8), 1415–1428 (2007)
31. Putinar, M.: Positive polynomials on compact semi-algebraic sets. Indiana Univ. Math. J. **42**(3), 969–984 (1993)
32. Roux, P., Voronin, Y., Sankaranarayanan, S.: Validating numerical semidefinite programming solvers for polynomial invariants. Formal Methods Syst. Des. **53**(2), 286–312 (2018)
33. Salamati, A., Zamani, M.: Data-driven safety verification of stochastic systems via barrier certificates: a wait-and-judge approach. In: Learning for Dynamics and Control Conference, L4DC 2022. Proceedings of Machine Learning Research, vol. 168, pp. 441–452. PMLR (2022)
34. Sassi, M.A.B., Girard, A., Sankaranarayanan, S.: Iterative computation of polyhedral invariants sets for polynomial dynamical systems. In: 53rd IEEE Conference on Decision and Control, CDC 2014, pp. 6348–6353. IEEE (2014)
35. Sassi, M.A.B., Sankaranarayanan, S., Chen, X., Ábrahám, E.: Linear relaxations of polynomial positivity for polynomial Lyapunov function synthesis. IMA J. Math. Control. Inf. **33**(3), 723–756 (2016)
36. Sogokon, A., Ghorbal, K., Johnson, T.T.: Non-linear continuous systems for safety verification. In: ARCH@CPSWeek 2016, 3rd International Workshop on Applied Verification for Continuous and Hybrid Systems. EPiC Series in Computing, vol. 43, pp. 42–51. EasyChair (2016)
37. Sogokon, A., Ghorbal, K., Tan, Y.K., Platzer, A.: Vector barrier certificates and comparison systems. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 418–437. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_25
38. Stone, M.H.: The generalized weierstrass approximation theorem. Math. Mag. **21**(5), 237–254 (1948). http://www.jstor.org/stable/3029337
39. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. University of California Press, Berkeley (May (1951)
40. Wang, J., Magron, V., Lasserre, J.: TSSOS: a moment-SOS hierarchy that exploits term sparsity. SIAM J. Optim. **31**(1), 30–58 (2021)
41. Wang, Q., Chen, M., Xue, B., Zhan, N., Katoen, J.: Encoding inductive invariants as barrier certificates: synthesis via difference-of-convex programming. Inf. Comput. **289**(Part), 104965 (2022)
42. Wang, Q., Li, Y., Xia, B., Zhan, N.: Generating semi-algebraic invariants for non-autonomous polynomial hybrid systems. J. Syst. Sci. Complexity **30**(1), 234–252 (2017)
43. Wongpiromsarn, T., Topcu, U., Lamperski, A.: Automata theory meets barrier certificates: temporal logic verification of nonlinear systems. IEEE Trans. Autom. Control **61**(11), 3344–3355 (2016)
44. Xu, X., Tabuada, P., Grizzle, J.W., Ames, A.D.: Robustness of control barrier functions for safety critical control. In: 5th IFAC Conference on Analysis and Design of Hybrid Systems, ADHS 2015. IFAC-PapersOnLine, vol. 48, pp. 54–61. Elsevier (2015)
45. Zeng, X., Lin, W., Yang, Z., Chen, X., Wang, L.: Darboux-type barrier certificates for safety verification of nonlinear hybrid systems. In: International Conference on Embedded Software, 2016, pp. 1–10. ACM (2016)
46. Zhao, H., Qi, N., Dehbi, L., Zeng, X., Yang, Z.: Formal synthesis of neural barrier certificates for continuous systems via counterexample guided learning. ACM Trans. Embed. Comput. Syst. **22**(5s), 1–21 (2023)

47. Zhao, H., Zeng, X., Chen, T., Liu, Z.: Synthesizing barrier certificates using neural networks. In: HSCC 2020: 23rd ACM International Conference on Hybrid Systems: Computation and Control, Sydney, New South Wales, Australia, April 21–24, 2020, pp. 1–11. ACM (2020)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Tolerance of Reinforcement Learning Controllers Against Deviations in Cyber Physical Systems

Changjian Zhang[1], Parv Kapoor[1], Rômulo Meira-Góes[2], David Garlan[1], Eunsuk Kang[1](✉), Akila Ganlath[3], Shatadal Mishra[3], and Nejib Ammar[3]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
{changjiz,parvk,dg4d,eunsukk}@andrew.cmu.edu
[2] the Pennsylvania State University, State College, PA, USA
romulo@psu.edu
[3] Toyota InfoTech Labs, Mountain View, CA, USA
{akila.ganlath,shatadal.mishra,nejib.ammar}@toyota.com

Abstract. Cyber-physical systems (CPS) with reinforcement learning (RL)-based controllers are increasingly being deployed in complex physical environments such as autonomous vehicles, the Internet-of-Things (IoT), and smart cities. An important property of a CPS is *tolerance*; i.e., its ability to function safely under possible disturbances and uncertainties in the actual operation. In this paper, we introduce a new, expressive notion of tolerance that describes how well a controller is capable of satisfying a desired system requirement, specified using *Signal Temporal Logic (STL)*, under possible *deviations* in the system. Based on this definition, we propose a novel analysis problem, called the *tolerance falsification problem*, which involves finding small deviations that result in a violation of the given requirement. We present a novel, two-layer simulation-based analysis framework and a novel search heuristic for finding small tolerance violations. To evaluate our approach, we construct a set of benchmark problems where system parameters can be configured to represent different types of uncertainties and disturbances in the system. Our evaluation shows that our falsification approach and heuristic can effectively find small tolerance violations.

1 Introduction

The *tolerance* of a CPS characterizes the ability of an engineered system to function correctly in the presence of uncertainties. Modern *cyber-physical systems (CPS)* operate in dynamic and uncertain environments, such as autonomous vehicles, medical devices, the Internet of Things (IoT), and smart cities. The mission-critical and safety-critical nature of CPS accentuate the need to provide a high level of tolerance against uncertainties, as a failure to do so could result in severe consequences, from safety hazards to economic losses.

C. Zhang, P. Kapoor—Both authors contributed equally to this research.

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 267–285, 2025.
https://doi.org/10.1007/978-3-031-71177-0_17

As CPS grow in complexity and scale, *reinforcement learning (RL)* techniques are gaining popularity for learning CPS controllers. In general, these controllers perceive the state of the CPS and take an action that maximizes the *long-term utility*. The utility is captured through reward functions designed by engineers. An RL controller is trained via a trial-and-error process where an agent takes actions in a simulator of the CPS and uses the simulated results of the actions to discover an optimal control strategy. Hence, the fidelity of the simulator plays a big role in the effectiveness of a trained controller. Often, there are reality gaps between the actual deployed environment and the simulator due to approximation and under-modeling of physical phenomena, which makes controllers trained in simulations perform poorly in the real world [1]. This performance degradation can also manifest as unsafe system behaviors in the actual environment.

To make an RL controller tolerant of possible errors due to these reality gaps, existing works often focus on the training stage, such as robust RL [2,3] and domain randomization [4–6]. They investigate the problem of training a controller that is capable of maintaining desired system behavior in the presence of possible *system deviations*—environmental uncertainties, observation or actuation errors, disturbances, and modeling errors. However, these methods are limited in how desired system behaviors are expressed. In RL, the desired behavior is often expressed using a reward function [2,3]; it is well-known that encoding a high-level system requirement using a reward function is a challenging task that requires a significant amount of domain expertise and manual effort via reward shaping [7,8]. Additionally, certain requirements cannot be directly encoded as rewards, especially those that capture time-varying behavior (e.g., "the vehicle must come to a stop in the next 3 s").

Due to the limitation in reward functions and the data-driven nature of RL, these training-oriented methods in general do not provide formal guarantees about tolerance. Also, there is a lack of focus on post-training analysis for the tolerance of RL controllers, especially in the sense of maintaining a desired, complex system specification. Moreover, a formal definition of tolerance for RL controllers with respect to system behavior (beyond rewards) is also missing.

To fill the missing gap in post-training tolerance analysis of RL controllers, we propose a new notion of tolerance based on specifications in *Signal Temporal Logic (STL)* [9]. Our definition assumes a *parametric* representation of a system, where *system parameters* capture the dynamics of the system (e.g., acceleration of a nearby vehicle) that are affected by system deviations (e.g., sensor errors). A system is initially assigned a set of nominal parameters that describe its expected dynamics. Then, a change in parameters, denoted by δ, corresponds to a deviation that may occur. Finally, a controller is said to be *tolerable* against certain deviations with respect to a STL specification if and only if the controller is capable of *satisfying* the specification even under those deviations.

Based on this tolerance definition, we propose a new type of analysis problem called the *tolerance falsification*. The goal is to find deviations in system parameters that result in a violation of the desired system specification. Specifically, we argue that identifying a violation closer to the nominal system parameters would be more valuable, since such a violation is more likely to occur in practice. Intuitively, our system needs to tolerate these deviations before addressing the ones that are further away from the nominal set. These identified violations could be used to retrain the controller for improved tolerance, or to build a run-time monitor to detect when the system deviates into an unsafe region.

In addition, we propose a novel simulation-based framework where the tolerance falsification problem is formulated as a two-layer optimization problem. In the lower layer, for a given system deviation δ (representing a particular system dynamics), an optimization-based method is used to find a falsifying signal; i.e., a sequence of system states that results in a violation of the given STL specification. In the upper layer, the space of possible deviations is explored to find small deviations that result in a specification violation, repeatedly invoking the lower-layer falsification. The results generated from the lower layer guide the upper-layer search towards small violating deviations. Furthermore, we present a novel heuristic that leverages the differences between the trajectories from the normative and deviated environments, captured via cosine distances, to improve the effectiveness of the upper layer search algorithm.

To evaluate the effectiveness of our falsification approach, we have constructed a set of benchmark case studies. In particular, these benchmark systems are configurable with system parameters to generate a range of systems with different behaviors due to the parameters' impact on how the system evolves. Our evaluation shows that our approach can be used to effectively find small deviations that cause a specification violation in these systems.

This paper makes the following contributions:

- We present a novel, formal definition of tolerance for RL controllers (Sect. 4), and a new analysis problem named *tolerance falsification problem* (Sect. 5).
- We propose a two-layer optimization-based algorithm and a novel search heuristic for finding small violating deviations (Sect. 6).
- We present an RL tolerance analysis benchmark and evaluate the effectiveness of our approach through experimental results on it (Sect. 7).

2 Preliminaries

Markov Decision Process. We model the systems under study as discrete-time stochastic systems in *Markov Decision Processes* (MDPs) [10]. An MDP is a tuple $\mathbf{M} = \langle S, A, T, I, R \rangle$, where $S \subseteq \mathbb{R}^n$ is the set of states, $A \subseteq \mathbb{R}^m$ is the set of actions (e.g., control inputs), $T : S \times A \times S \to [0, 1]$ is the transition function where $T(s, a, s')$ represents the probability from state s to s' by action a and $\forall s \in S, a \in A : \sum_{s' \in S} T(s, a, s') = 1$, $I : S \to [0, 1]$ is the initial state distribution, and $R : S \to \mathbb{R}$ is the reward function. As is often the case for real-world systems, we assume that the transition function is unknown.

We consider black-box deterministic control policies for a system. Formally, a policy $\pi : S \to A$ for an MDP maps states to actions. Reinforcement learning (RL) [11] is the process of learning an optimal policy π^* that maximizes the cumulative discounted reward for this MDP. Additionally, a trajectory σ of an

MDP given an initial state $s_0 \sim I$ and a policy π is defined accordingly as $\sigma = (s_0 \xrightarrow{a_0} s_1 \ldots s_i \xrightarrow{a_i} s_{i+1} \ldots)$ where $a_i = \pi(s_i)$ and $s_{i+1} \sim T(s_i, a_i)$. Finally, we use $\mathcal{L}(\mathbf{M}||\pi)$ to represent the behavior of the controlled system, i.e., it is the set of all trajectories of a system $\mathbf{M}$ under the control of π.

Signal Temporal Logic. A signal $\mathbf{s}$ is a function $\mathbf{s} : T \to D$ that maps a time domain $T \subseteq \mathbb{R}_{\geq 0}$ to a k real-value space $D \subseteq \mathbb{R}^k$, where $\mathbf{s}(t) = (v_1, \ldots, v_k)$ represents the value of the signal at time t. Then, an STL formula is defined as:

$$\phi := \mu \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi\, \mathcal{U}_{[a,b]}\, \psi$$

where μ is a predicate of the signal $\mathbf{s}$ at time t in the form of $\mu \equiv \mu(\mathbf{s}(t)) > 0$ and $[a, b]$ is the time interval (or simply I). The *until* operator $\mathcal{U}$ defines that ϕ must be true until ψ becomes true within a time interval $[a, b]$. Two other operators can be derived from *until*: *eventually* ($\Diamond_{[a,b]}\ \phi := \top\ \mathcal{U}_{[a,b]}\ \phi$) and *always* ($\Box_{[a,b]}\ \phi := \neg\Diamond_{[a,b]}\ \neg\phi$).

The satisfaction of an STL formula can be measured in a quantitative way as a real-valued function $\rho(\phi, \mathbf{s}, t)$ (also known as the STL *robustness* value), which represents the difference between the actual signal value and the expected one [9]. For example, given a formula $\phi \equiv \mathbf{s}(t) - 3 > 0$, if $\mathbf{s} = 5$ at time t, then the satisfaction of ϕ can be evaluated by $\rho(\phi, \mathbf{s}, t) = \mathbf{s}(t) - 3 = 2$. The definition of ρ is as follows (ρ for the other operators can be formulated from these):

$$\begin{aligned}
&\rho(\mu, \mathbf{s}, t) = \mu(\mathbf{s}(t)) \qquad\qquad \rho(\neg\phi, \mathbf{s}, t) = -\rho(\phi, \mathbf{s}, t)\\
&\rho(\phi \wedge \psi, \mathbf{s}, t) = \min\{\rho(\phi, \mathbf{s}, t), \rho(\psi, \mathbf{s}, t)\}\\
&\rho(\phi\, \mathcal{U}_I\, \psi, \mathbf{s}, t) = \sup_{t_1 \in I+t} \min\{\rho(\psi, \mathbf{s}, t_1), \inf_{t_2 \in [t, t_1]} \rho(\phi, \mathbf{s}, t_2)\}
\end{aligned}$$

3 Motivating Example

We use an RL system which is required to satisfy a safety specification to illustrate our tolerance definition and analysis. Consider the *CarRun* safe RL system implemented in bullet-safety-gym[1], depicted in Fig. 1. The CarRun system has a four-wheeled agent based on *MIT Racecar*[2] placed between two safety boundaries. The safety boundaries are non-physical bodies that can be breached without causing a collision. The objective is to go through the avenue between the boundaries without penetrating them. The agent velocity also needs to be maintained below a user-defined threshold. Formally, it can be specified by an STL invariant: $\Box(|y_{pos}| < C_1 \wedge |v| < C_2)$, where C_1 and C_2 are the constant thresholds for the y coordinate and the velocity, respectively.

Given the CarRun system, we can train an RL controller such that the car agent satisfies the safety specification above using methods from safe RL [12] [13]. However, to transfer this "safe" controller to the real world, we need to account

[1] https://github.com/SvenGronauer/Bullet-Safety-Gym.
[2] https://github.com/mit-racecar.

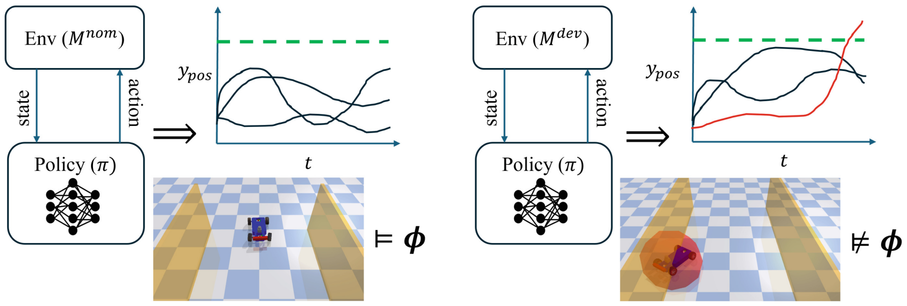


Fig. 1. Behavior of the CarRun system under different system parameters. In the norminal condition (left), y_{pos} in all trajectories is below the threshold (green line) and thus the system is safe. However, in the deviated condition (right), there exists a trajectory where y_{pos} exceeds the threshold and hence the safety requirement is violated.

for the reality gap between the simulator and the deployed environment. This reality gap might arise due to inaccurate modeling of contact surfaces, actuator errors, and incorrect physical parameter configuration (e.g., friction and mass). These reality gaps can lead to the agent violating the safety specification in the real world, despite satisfying them in simulation. Additionally, since the RL controllers are black-box neural networks, it is extremely hard to capture their concrete behaviors. The difficulty in reasoning about the controller's behaviors coupled with the stochasticity of the system leads to a challenging analysis problem of understanding their tolerance ability. This has long been one of the key drawbacks that limit the application of these controllers in the real world [6,14].

Since it can be challenging to quantitatively measure these reality gaps, we take a parametric approach. We approximate the reality gap between the simulator and the deployed environment quantitatively using deviations as parameters. For example, we model the CarRun system as being *parametric* with two controllable *system parameters*, tm (turn multiplier, a factor for the steering control) and sm (speed multiplier, a factor for the speed control). These parameters govern the impact of the action provided by the controller, e.g., a larger sm will result in more aggressive accelerations. The intuition behind these deviations is to account for actuation issues that arise while deploying agents in the real world. Figure 1 shows the behavior of CarRun under different system parameters. In Fig. 1(a), the agent is deployed in the nominal condition with default system parameters. In this scenario, the controller successfully manages to drive the Car agent through the avenue and also maintains a safe velocity, i.e., the safety specification is satisfied. In Fig. 1(b), we show the same controller deployed under a deviated CarRun environment with different turn and speed multipliers. In this scenario, the controller makes the car behave erratically, which eventually makes the car cross the safety boundary, i.e., the safety specification is violated.

This example highlights the brittleness of these controllers concerning safety specifications and the need for stakeholders to address pre-deployment questions like: *What are the possible deviations that these RL controllers can tolerate?* More specifically, *how much change in the system parameters can the controller tolerate before it begins to violate the given safety specification?* We formulate this question as a type of analysis problem called *tolerance falsification*, where the goal is to find deviations in system parameters (e.g., the changes in the turn and speed multiplier of CarRun) where the deviated system violates the given specification. This analysis problem is challenging due to the stochastic, black-box nature of the system as well as the opacity of NN-based RL controllers.

Additionally, a notion of "quality of solution" while searching for system parameters is necessary to factor in the practical assumptions about the operating context of this system. For example, deviations that are closer to the nominal parameters are more likely to occur in practice and hence need to be prioritized when analyzing. This helps avoid impractically large deviation values that might cause a violation but offers little insight to system designers. Thus, our falsification process attempts to find violations with *small* deviations; i.e., minimal parameter changes that introduce a risk of specification violation into the system. The output of this analysis (i.e., violations) can help the engineer identify RL-based controller brittleness and can be used to redesign or retrain the controller to improve its tolerance.

4 Tolerance Definition

4.1 Definition of Specification-Based Tolerance

In this work, we use STL to specify the desired properties of a system, and system parameters to capture the deviations in system dynamics. Parameters can represent a variety of deviations such as environmental disturbances (e.g., wind or turbulence), internal deviations (e.g., mass variation of a vehicle), observation errors (e.g., sensor errors), or actuation errors (e.g., errors in steering control). Then, to capture systems with such diverse dynamics using parameters, we leverage the notion of *parametric control systems* [15,16].

A *parametric* discrete-time stochastic system $\mathbf{M}^{\Delta}$ defines a set of systems such that $\Delta \subseteq \mathbb{R}^k$ represents the parameter domain, and for any $\delta \in \Delta$, an instance of a parametric system $\mathbf{M}^{\delta}$ is an MDP $\mathbf{M}^{\delta} = \langle S, A, T^{\delta}, I^{\delta}, R \rangle$, where the initial state distribution I^{δ} and the state transition distributions T^{δ} are both defined by the parameter δ. Parameter δ represents a deviation to a system and Δ represents the domain of all deviations of interest. In addition, we use $\delta_0 \in \Delta$ to represent the zero-deviation point, i.e., the parameter under which the system $\mathbf{M}^{\delta_0}$ exhibits the expected, normative behavior. Then, we define a system as being tolerable against a certain deviation as follows:

Definition 1. *For a system $\mathbf{M}$, a policy π, a deviation parameter δ, and an STL property ϕ, we say the system can* tolerate *the deviation when the parametric form of $\mathbf{M}$ with δ under the control of π satisfies the property, i.e., $\mathbf{M}^{\delta} || \pi \models \phi$.*

Then, the tolerance of a controller can be defined as all the possible deviations that the system can tolerate. Formally:

Definition 2. *For a system* $\mathbf{M}$*, a policy* π*, and an STL property* ϕ*, the tolerance of the controller is defined as the* ***maximal*** $\boldsymbol{\Delta} \subseteq \mathbb{R}^k$ *s.t.* $\forall \delta \in \boldsymbol{\Delta} : \mathbf{M}^\delta || \pi \models \phi$*.*

In other words, the tolerance of a control policy π is measured by the maximal parameter domain $\boldsymbol{\Delta}$ of a system where each deviated system $\mathbf{M}^\delta$ of it still satisfies the property under the control of π.

4.2 Strict Evaluation of Tolerance

In this work, we focus on a specific evaluation of tolerance. Specifically, Def. 1 and 2 depend on the interpretation of $\mathbf{M}^\delta || \pi \models \phi$, i.e., a system satisfying a STL property; however, STL satisfaction is computed over a single trajectory. From the literature [17], one common evaluation criteria is that *a system must not contain a trajectory that violates the STL property.* In other words, even in the worst-case scenario that is less likely to occur in a stochastic system, it should still guarantee the property. This interpretation enforces a strong guarantee of the system, and thus we call it the *strict* satisfaction of STL in this work. Formally:

Definition 3. *A discrete-time stochastic system* $\mathbf{M}$ strictly *satisfies an STL property* ϕ *under the control of a policy* π *iff every controlled trajectory produces a non-negative STL robustness value, i.e.,* $\mathbf{M} || \pi \models \phi \Leftrightarrow \forall \sigma \in \mathcal{L}(\mathbf{M} || \pi) : \rho(\phi, \mathbf{s}_\sigma, 0) \geq 0$*, where* $\mathbf{s}_\sigma$ *is the signal of state values of trajectory* σ*.*

With this interpretation, we can then restate Def. 2 as:

Definition 4. *The tolerance of a policy* π *that* strictly *satisfies an STL property* ϕ *is the maximal* $\boldsymbol{\Delta}$ *s.t.* $\forall \delta \in \boldsymbol{\Delta}, \sigma \in \mathcal{L}(\mathbf{M}^\delta || \pi) : \rho(\phi, \mathbf{s}_\sigma, 0) \geq 0$

Although this definition delineates a strong tolerance guarantee, it can also be extended to more relaxed notions with probabilistic guarantees. In that case, other evaluation techniques for STL specification satisfaction such as [18–20] can be leveraged. We leave this as an extension of our work in the future.

5 Tolerance Analysis

5.1 Tolerance Falsification

According to Def. 4, to compute the tolerance of a controller, we need to: (1) (formally) show that a stochastic system does not contain a trajectory that violates the STL property, and (2) compute the maximal parameter set $\boldsymbol{\Delta}$, which could be in any non-convex or even non-continuous shape, where all system instances $\mathbf{M}^\delta$ should satisfy step (1). This exhaustive computation is intractable due to the black-box RL controllers coupled with the stochasticity in system.

Therefore, in this work, instead of computing or approximating the tolerance $\boldsymbol{\Delta}$, we consider the problem of falsifying a given estimation of tolerance $\widehat{\boldsymbol{\Delta}}$, i.e., finding a deviation $\delta \in \widehat{\boldsymbol{\Delta}}$ that the system cannot tolerate for a given controller.

Problem 1 (Tolerance Falsification). For a system $\mathbf{M}$, a policy π, and an STL property ϕ, given a tolerance estimation $\widehat{\mathbf{\Delta}} \subseteq \mathbb{R}^k$, the goal of a *tolerance falsification problem* $\mathcal{F}(\mathbf{M}, \pi, \phi, \widehat{\mathbf{\Delta}})$ is to find a deviation $\delta \in \widehat{\mathbf{\Delta}}$ s.t. $\exists \sigma \in \mathcal{L}(\mathbf{M}^\delta || \pi) : \rho(\phi, \mathbf{s}_\sigma, 0) < 0$.

5.2 Minimum Tolerance Falsification

Intuitively, a larger deviation (i.e., a deviation that is far away from the expected system parameter) would likely cause a larger deviation in the system behavior leading to a specification violation. However, controllers are generally not designed to handle arbitrarily large deviations in the first place, and analyzing their performance in these situations offers limited insight to the designer. Moreover, if the designer decides to improve the tolerance of a controller (which is a costly endeavor), deviations closer to the nominal system are given high priority due to their higher likelihood of occurrence. In light of these practical design and deployment assumptions, we focus on the *minimum* deviation problem.

Problem 2. Given a *minimum* tolerance falsification problem $\mathcal{F}_{min}(\mathbf{M}, \pi, \phi, \widehat{\mathbf{\Delta}})$, let $\delta_0 \in \widehat{\mathbf{\Delta}}$ be the zero-deviation point, the goal is to find a deviation $\delta \in \widehat{\mathbf{\Delta}}$ s.t. $\mathbf{M}^\delta || \pi \not\models \phi$ and δ minimizes a distance measure $\|\delta - \delta_0\|_p$.

5.3 Falsification by Optimization

Since the satisfaction of STL can be measured quantitatively, the tolerance falsification problem can be formulated as an optimization problem. Consider a real-valued system evaluation function $\Gamma(\mathbf{M}, \pi, \phi) \in \mathbb{R}$. We assume that if this function's value is negative, the controlled system violates the property, i.e., $\Gamma(\mathbf{M}, \pi, \phi) < 0 \Leftrightarrow \mathbf{S} || \pi \not\models \phi$, and the smaller the value, the larger the degree of property violation. Then, a tolerance falsification problem $\mathcal{F}(\mathbf{M}, \pi, \phi, \widehat{\mathbf{\Delta}})$ can be formulated as the following optimization problem:

$$\underset{\delta \in \widehat{\mathbf{\Delta}}}{\arg\min}\ \Gamma(\mathbf{M}^\delta, \pi, \phi) \tag{1}$$

i.e., by finding a parameter $\delta \in \widehat{\mathbf{\Delta}}$ that minimizes the evaluation function Γ and observing this value can give information about system's property satisfaction. Concretely, if the minimum function value is negative, then the associated parameter δ indicates a deviation where the system violates the property ϕ. Specifically, in the case of strict evaluation of tolerance, the system evaluation function Γ is defined as:

$$\Gamma(\mathbf{M}, \pi, \phi) = \min\{\rho(\phi, \mathbf{s}_\sigma, 0) \mid \sigma \in \mathcal{L}(\mathbf{M} || \pi)\} \tag{2}$$

Note that Eq. 2 is a typical formulation for solving a CPS falsification problem that intends to find a trajectory that violates an STL specification [17].

Finally, we can formulate a minimum tolerance falsification problem $\mathcal{F}_{min}(\mathbf{M}, \pi, \phi, \widehat{\boldsymbol{\Delta}})$ as a constrained optimization problem:

$$\underset{\delta \in \widehat{\boldsymbol{\Delta}}}{\arg\min} \ \|\delta - \delta_0\|_p \ s.t. \ \Gamma(\mathbf{M}^\delta, \pi, \phi) < 0 \tag{3}$$

Equation 1 and 3 can both be seen as a bi-level optimization problem [21]; the upper-layer task searches for deviation parameters (δ) and the lower-layer searches for system trajectories. The problem of finding *any* tolerance violation (Eq. 1) can also be formulated as a min-min optimization problem, which can be solved by existing CPS falsifiers such as Breach [22] and PsyTaLiRo [23,24].

However, the *minimum* falsification problem (Eq. 3) features multi-objective optimization or min-max optimization characteristics—minimizing the deviation distance ($\|\delta - \delta_0\|_p$) would likely cause a larger system evaluation value (Γ). Since these objectives are inherently conflicting, nuanced techniques are required to find solutions. Although, existing CPS falsifiers can be configured to represent this additional cost/objective function (either via specification modification or through explicit cost function definition), the underlying optimization techniques do not have a multi-layer setup to handle this off the shelf. Therefore, we present a novel two-layer search for solving the tolerance falsification problems, particularly effective in finding minimum violating deviations.

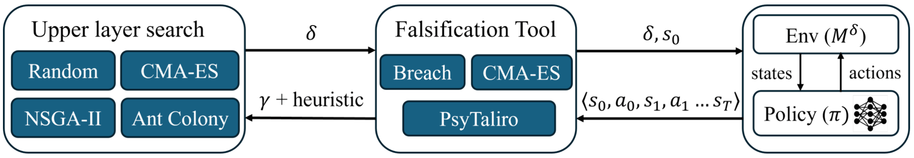


Fig. 2. Overview of the two-layer falsification algorithm.

6 Simulation-Based Tolerance Analysis Framework

In this section, we outline our analysis framework to solve the tolerance falsification problems for black-box CPS and RL controllers (as shown in Fig. 2). We first explain our novel two layer falsification algorithm and then present a heuristic for more effective solving of the minimum falsification problem.

6.1 A Two-Layer Falsification Algorithm

Algorithm 1 describes our two-layer framework. Lines 3–13 indicate the upper-layer search. In each iteration, the upper-layer searches a set of deviation samples. For a deviation δ, it instantiates a deviated system $\mathbf{M}^\delta$ (line 6), computes the system evaluation value γ (line 7), and then computes the objective function

Algorithm 1: A Two-Layer Tolerance Falsification Algorithm

```
Input  : $\mathbf{M}, \pi, \phi, \widehat{\mathbf{\Delta}}$, and objective function $f$
Output: violation $\delta_{best} \in \widehat{\mathbf{\Delta}}$
1  $\delta_{best} \leftarrow nil$;
2  $X \leftarrow$ initial candidates from $\widehat{\mathbf{\Delta}}$ ;
3  while termination criteria = false do
4      $V \leftarrow \langle\rangle$ ;
5      for $\delta \in X$ do
6          $M^{\delta} \leftarrow$ Instantiate($\mathbf{M}^{\widehat{\mathbf{\Delta}}}, \delta$) ;
7          $\gamma \leftarrow$ CPSFalsification($\mathbf{M}^{\delta}, \pi, \phi$) ;
8          $v \leftarrow f(\delta, \gamma)$ ;                    // heuristic computation.
9          $V \leftarrow V \frown \langle v \rangle$ ;
10     end
11     $\delta_{best} \leftarrow$ UpdateBest($X$, $V$) ;
12     $X \leftarrow$ NextCandidates($f, X, V, \widehat{\mathbf{\Delta}}$) ;
13 end
```

value v (line 8). The objective value indicates the quality of a deviation sample, e.g., whether it causes a violation and has a small distance to the zero-deviation point. Finally, the objective values are used to update the best result so far (line 11) and generates the next candidate solutions (line 12). In particular, line 7 indicates the lower-layer task. It corresponds to the system evaluation function Γ (which is the minimal STL robustness value according to Eq. 2).

Given the characteristics of our falsification problem, we propose this two-layer structure for multiple reasons: First, the separation of deviations and the lower-layer CPS falsification allows us to define richer evaluation metrics and heuristics that are solely relevant for deviation searching. These heuristics, if used in a single layer objective, would lead to an ill-posed optimization problem exacerbated by the highly non-convex landscapes of traditional CPS falsification. Second, this separation of concerns allow us to find deviations closer to nominal points even for systems with high-dimensional state spaces, complex dynamics, and rugged robustness landscapes with multiple local minimas. In these settings, an one-layer search would converge to local solutions without exploring the search space extensively. Finally, this two-layer structure provides us enough extensibility to:

- Integrate many off-the-shelf optimization methods for the upper-layer like we have for Uniform Random, CMA-ES [25], NSGA-II [26], and Ant Colony [27].
- Integrate state-of-the-art CPS falsifiers (we integrated CMA-ES, Breach [22], and PsyTaLiRo [24]) and simulation platforms (we used OpenAI-Gym [28], PyBullet [29], and Matlab Simulink).
- Extend to other STL evaluation methods (function Γ), e.g., evaluation with probabilistic guarantees [18–20], cumulative STL [30], or mean STL [31].

6.2 Heuristic for Efficient Minimum Tolerance Falsification

We present a novel heuristic for more effective discovery of minimum violating deviations. Our heuristic is based on the known issues of RL policy overfitting. It has been highlighted in related literature that RL policies can overfit to the specific paramterized system used for training and this dependence can reduce their applicability to real-world scenarios [4–6]. We exploit this over-fitting tendency to guide the search for δ that leads to a violation. Our heuristic is the cosine similarity between a deviated system's worst-case trajectory and a nominal system's worst-case trajectory. Formally:

$$dist(\delta) = \frac{\mathbf{Tr}_\delta \cdot \mathbf{Tr}_{\delta_0}}{\|\mathbf{Tr}_\delta\| \cdot \|\mathbf{Tr}_{\delta_0}\|}$$

Specifically, when computing the objective function value v (line 8), we add the similarity value $dist(\delta)$ to the system evaluation value γ. Our intuition is that once a controller has been trained in a system parameterized by δ_0, it overfits to that specific system. Then, when the controller is deployed in a deviated system, its worst-case trajectory will be similar to the nominal worst-case trajectory if the distance between the two MDPs, measured by the Euclidean distance between the parameters, is small. We measure the similarity between trajectories using cosine similarity. Thus, as the distance from the nominal MDP increases, the similarity score between the worst-case trajectories decreases. This heuristic provides more information about the search space: i.e. in the case there are two deviations where the robustness values are similar (which is possible due to the worst case semantics of STL robustness), cosine similarity can help in directing the search toward more violating directions.

A more in-depth discussion of this heuristic, along with an example, can be found in the extended version of this paper [32].

7 Evaluation

We implemented our proposed framework in a Python package[3] and evaluate our technique through comprehensive experimentation. Our evaluation focuses on the minimum tolerance falsification problem. Specifically, we measure our technique's effectiveness through three key metrics: (1) *the number of violations found*, (2) *the minimum distance of violations*, and (3) *the average distance of violations*. Based on these metrics, we formulate the following research questions:

- **RQ1**: Is our two-layer falsification framework more effective than leveraging an existing CPS falsifier?
- **RQ2**: Does our heuristic improve the effectiveness for finding minimum violating deviations, compared to off-the-self optimization algorithms?

[3] https://github.com/SteveZhangBit/STL-Robustness.

Although existing CPS falsifiers [22–24] cannot directly solve our minimum tolerance falsification problem (Problem 2), they allow customizing the objective function to optimize for both the deviation distance and STL robustness value to find minimum deviations. We call this technique *one-layer search*. For RQ1, we benchmark against the one-layer search baseline for the minimum tolerance falsification problem. For RQ2, we evaluate whether our proposed heuristic described in Sect. 6.2 further improves the effectiveness of our two-layer search, specifically the minimum distance.

7.1 Experimental Setup and Implementational Details

To answer these research questions, we first present a benchmark with systems and controllers trained to satisfy complex safety specifications. The benchmark contains six systems with non-linear dynamics adopted from OpenAI-Gym, PyBullet, and Matlab Simulink. We extend the interfaces of these systems so that users can configure their behavior for tolerance analysis by changing the system parameters. Details about the benchmark problems can be found in the extended version of this paper [32].

Then, we solve the corresponding minimum tolerance falsification problems for them. For each problem, we conduct the following experiments:

- One-layer search leveraging an existing CPS falsifier by modifying the objective function to factor in the deviation distance and STL robustness value,
- Two-layer search with CMA-ES for both the upper and lower layers,
- Two-layer search with CMA-ES+Heuristic for the upper layer and CMA-ES for the lower layer.

Specifically, for the one-layer search, we employ the state-of-the-art CPS falsifiers, Breach [22] (with CMA-ES) and PsyTaLiRo [24] (with dual annealing). We replace their default objective functions with the sum of the normalized deviation distance and STL robustness value.

For the two-layer search, due to the complexity of the CPS and the non-convex nature of STL robustness, the upper-layer optimization is also non-convex and has multiple local minima. Additionally, we assume black-box systems and controllers. Thus, due to these two considerations, we made the decision to adopt derivative-free evolutionary algorithms. Specifically, we primarily utilized CMA-ES as the upper-layer algorithm because it is widely used for black-box optimization and in our preliminary experiments outperformed other evolutionary methods. However, other algorithms can also be integrated. Furthermore, we also use CMA-ES for the lower-layer search as it is a widely used in CPS falsification tools [17,22] and works competitively for both Python and Matlab environments. Finally, we implement our heuristic and use it alongside the evaluation function for the upper-layer search.

Each problem was run three times on a Linux machine with a 3.6 GHz CPU and 24 GB memory. For fair evaluation, we set the budget in terms of the number of interactions with the simulator for all our techniques. Specifically, for one run, the budget for the one-layer search is 10,000 simulations; and the budget for the two-layer search is 100 for the upper-layer and 100 for the lower-layer falsification.

7.2 Results

Table 1 summarizes the results for solving the minimum tolerance falsification problems. The *Viol.* column shows the number of violations found in total from the three runs. The *Min Dst.* and *Avg. Dst.* columns show the minimum and average normalized l-2 distance to the zero-deviation point (i.e., $\|\delta - \delta_0\|_2$) of the found violations, respectively. The performance of our approach heavily depends

Table 1. Minimum tolerance falsification results.

	One-layer search			CMA-ES			CMA-ES w/ Heuristic		
	Viol.	Min Dst.	Avg. Dst.	Viol.	Min Dst.	Avg. Dst.	Viol.	Min Dst.	Avg. Dst.
Cartpole	**90**	0.300	**0.399**	69	0.285	0.449	79	**0.256**	0.417
LunarLander	–	–	–	74	0.026	**0.222**	**84**	**0.020**	0.293
CarCircle	11	0.143	0.255	22	0.102	**0.219**	**57**	**0.068**	0.454
CarRun	25	0.191	**0.249**	68	0.161	0.449	**109**	**0.156**	0.399
ACC	N/A	N/A	N/A	43	**0.110**	**0.323**	**110**	0.138	0.415
WTK	**300**	0.299	**0.443**	54	**0.296**	0.454	45	0.319	0.533

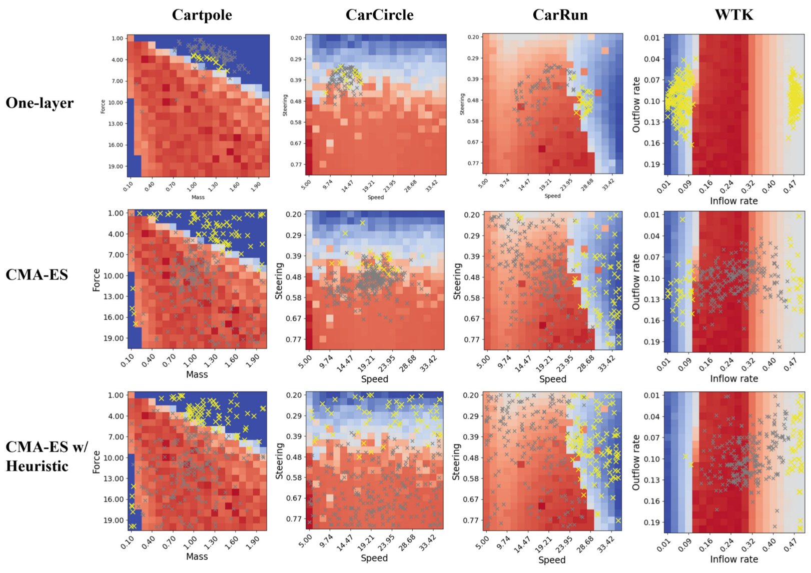


Fig. 3. Search spaces, deviation samples and violations processed by each algorithm. In each graph, the axes are the parameter domains. A red cell is a positive STL robustness value and a blue cell is a negative value. A grey cross is a deviation sample that is not falsified in the given budget; a yellow cross is a violation. (Color figure online)

on the underlying simulation time of a system that vastly outweighs the overhead added by our evolutionary search algorithms. Thus, we share comparable performance, measured by total run time, as tools like Breach and PsyTaLiRo given the same budget of simulation calls.

In addition, to qualitatively exhibit our approach's effectiveness in finding deviations, we visualize the search space landscape for different problems in heat maps. Each heat map is generated by slicing the space (i.e., the estimated domain of system parameters) into a 20×20 grid and using a CPS falsifier to find the minimum STL robustness value for each grid cell. However, this processing is only done for visualization purposes and is not used in any of the algorithms. This brute force sampling requires far more resources than our falsification approach. Finally, we draw the deviation samples and violations from our analysis on the heat maps. The final results are illustrated visually in Fig. 3.

Answer to RQ1. From the table, the one-layer search fails to find violations in LunarLander, and it cannot represent the type of system parameters we need in ACC (due to falsification tool implementation). On the other hand, our two-layer search with CMA-ES solves all the problems and finds smaller deviations than the one-layer search in all problems. Moreover, from the heat maps, since the distance value is directly added to the STL robustness value in the one-layer search, it fails to find small deviations that *barely* violate the property because it would result in a larger objective value. Thus, it is hard for it to converge to the minimum violating deviations. On the other hand, our two-layer search can better converge to the boundary of safe and unsafe regions. However, it also causes it to find fewer violations because it searches for more samples in the safe region close to the boundary where violations can be rare.

Answer to RQ2. Our two-layer search with CMA-ES+Heuristic finds smaller violating deviations than the original CMA-ES in 4/6 problems. It also finds more violations in 5/6 problems. However, the average distances also increase in 4/6 problems due to more exploration of violations encouraged by our heuristic. Despite that, from the heat maps, our CMA-ES+Heuristic approach can still converge to small violating deviations on the safe and unsafe boundary while also finding more violations. Our heuristic helps in guiding the search and provides additional information to the algorithm when STL robustness is not enough to provide directionality. Concretely, a small similarity value would likely lead to a violation (even when the robustness value is similar) and thus results in more violations found and faster convergence to a small violation.

8 Related Work

There exists similar CPS tolerance notions from a control theory perspective such as [33,34]. For example, Saoud et al. [33] present a resilience notion of CPS based on LTL w.r.t. a real-valued disturbance space, which forms a ball around a particular trajectory. Then, they present a method to approximate the maximum set of disturbances that maintain a desired LTL property for linear control systems. These notions target traditional controllers with a white-box assumption of systems and controllers, whereas we employ a black-box assumption which is more practical regarding complex CPS and NN-based RL controllers.

Falsification of CPS [17] is a well-studied problem in the literature. It finds trajectories that violate a STL property by mutating the initial states or system inputs. A related application is parameter synthesis [35] that finds system parameters where the system satisfies the property. It can be seen as a dual problem to the falsification problem. Tools like Breach [22] and PSY-TaLiRo [23,24] support both types of analysis. However, our tolerance falsification problem can be seen as solving these two problems at the same time. Our upper-layer search finds system parameters that lead to a violation of the system specification, and the lower-layer search finds initial states or system inputs that lead to a violating trajectory. Although our problem can be reduced to a CPS falsification problem with system parameters, it is not effective in solving our minimum tolerance falsification problem compared to our two-layer structure.

A two-layer optimization structure has also been applied in CPS falsification such as in [36–38]. However, these approaches still target traditional falsification of a CPS (the lower-layer in our case), whereas our approach aims to separate the problems of finding small deviation parameters from system falsification.

VerifAI [39,40] applies a similar idea to us where they consider *abstract features* for a ML model that can lead to a property violation of a CPS. Different from us, they assume a CPS with a ML perception model (such as object detection) connecting to a traditional controller, and the abstract features are environmental parameters that would affect the performance of the ML model (e.g., brightness). In other words, they focus on deviations that affect the ML model whereas our deviation notion is more general that includes any external or internal deviation or sensor error which changes the system dynamics.

Robust RL studies the problem to improve the performance of controllers in the presence of uncertainties [2,3]. Also, domain randomization [4–6] studies how to train a controller that works across various systems with randomized parameters. However, our work is different in that: (1) we focus on tolerance evaluation whereas they focus more on training; and (2) we focus on system specifications in STL properties, while they rely on rewards where maximizing the reward does not necessarily guarantee certain system specification.

9 Conclusion

In this paper, we have introduced a specification-based tolerance definition for CPS. This definition yields a new type of analysis problem, called *tolerance falsification*, where the goal is to find small changes to the system dynamics that result in a violation of a given STL specification. We have also presented a novel optimization-based approach to solve the problem and evaluated the effectiveness of it over our proposed CPS tolerance analysis benchmark.

Since our analysis framework is extensible, as part of future work, we plan to explore and integrate other types of evaluation functions Γ (e.g., evaluation with probabilistic guarantees [18–20]), different semantics of STL robustness (e.g., cumulative robustness [30]), or leveraging decomposition of STL for more effective falsification of complex specifications [41]. Moreover, we currently use l-2 norm to compute the deviation distances. In the future, we also plan to explore other distance notions such as Wasserstein Distance [42–44], which computes distribution distance between system dynamics.

Acknowledgements. We are grateful to our anonymous reviewers for their comments and Georgios Fainekos for a discussion on an earlier version of this paper. This research was sponsored by InfoTech Labs, Toyota Motor North America. This work was also supported in part by the NSF awards 2144860 and 2319317, and the NSA grant H98230-23-C-0274. Any views, opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the organizations.

Data Availability Statement. The source code of our tool and all the experimental results are available at the following URL: https://doi.org/10.5281/zenodo.12144853.

References

1. Collins, J.J., Howard, D., Leitner, J.: Quantifying the reality gap in robotic manipulation tasks. 2019 International Conference on Robotics and Automation (ICRA), pp. 6706–6712, (2018). https://api.semanticscholar.org/CorpusID:53208962
2. Moos, J., Hansel, K., Abdulsamad, H., Stark, S., Clever, D., Peters, J.: Robust reinforcement learning: a review of foundations and recent advances. Machine Learning and Knowledge Extraction, vol. 4, no. 1, pp. 276–315 (2022). https://www.mdpi.com/2504-4990/4/1/13
3. Xu, M., et al.: Trustworthy reinforcement learning against intrinsic vulnerabilities: robustness, safety, and generalizability (2022)
4. Peng, X.B., Andrychowicz, M., Zaremba, W., Abbeel, P.: Sim-to-real transfer of robotic control with dynamics randomization. In: 2018 IEEE International Conference on Robotics and Automation (ICRA), pp. 3803–3810 (2018)
5. Sadeghi, F., Levine, S.: CAD2RL: real single-image flight without a single real image (2017)
6. Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., Abbeel, P.: Domain randomization for transferring deep neural networks from simulation to the real world. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 23–30 (2017)
7. Ng, A.Y., Harada, D., Russell, S.J.: Policy invariance under reward transformations: theory and application to reward shaping. In: Proceedings of the Sixteenth International Conference on Machine Learning, ser. ICML 1999, pp. 278–287. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. (1999)
8. Booth, S., Knox, W.B., Shah, J., Niekum, S., Stone, P., Allievi, A.: The perils of trial-and-error reward design: Misdesign through overfitting and invalid task specifications. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 37, no. 5, pp. 5920–5929 (2023). https://ojs.aaai.org/index.php/AAAI/article/view/25733

9. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15297-9_9
10. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model Checking Probabilistic Systems, pp. 963–999. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_28
11. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT press (2018)
12. García, J., Fernández, F.: A comprehensive survey on safe reinforcement learning. J. Mach. Learn. Res. **16**(1), 1437–1480 (2015)
13. Gu, S.:et al.: A review of safe reinforcement learning: methods, theory and applications. arXiv, vol. abs/2205.10330, (2022). https://api.semanticscholar.org/CorpusID:248965265
14. Yu, W., Liu, C.K., Turk, G.: Policy transfer with strategy optimization. In: International Conference on Learning Representations (2019). https://openreview.net/forum?id=H1g6osRcFQ
15. Bhattacharyya, S.P., Chapellat, H., Keel, L.H.: Robust Control: The Parametric Approach, 1st edn. Prentice Hall PTR, USA (1995)
16. Weinmann, A.: Uncertain Models and Robust Control. Springer, Vienna(2012)
17. Corso, A., Moss, R., Koren, M., Lee, R., Kochenderfer, M.: A survey of algorithms for black-box safety validation of cyber-physical systems. J. Artif. Intell. Res. **72**, 377–428 (2021)
18. Fan, C., Qin, X., Xia, Y., Zutshi, A., Deshmukh, J.: Statistical verification of autonomous systems using surrogate models and conformal inference (2021)
19. Pedrielli, G., et al.: Part-X: a family of stochastic algorithms for search-based test generation with probabilistic guarantees. IEEE Trans. Autom. Sci. Eng. **21**(3), 4504–4525 (2024). https://doi.org/10.1109/TASE.2023.3297984
20. Lindemann, L., Matni, N., Pappas, G.J.: STL robustness risk over discrete-time stochastic processes. In: 2021 60th IEEE Conference on Decision and Control (CDC), pp. 1329–1335 (2021)
21. Colson, B., Marcotte, P., Savard, G.: An overview of bilevel optimization. Ann. Oper. Res. **153**, 235–256 (2007)
22. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_17
23. Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TaLiRo: a tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 254–257. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_21
24. Thibeault, Q., Anderson, J., Chandratre, A., Pedrielli, G., Fainekos, G.: PSY-TaLiRo: a python toolbox for search-based test generation for cyber-physical systems. In: Lluch Lafuente, A., Mavridou, A. (eds.) FMICS 2021. LNCS, vol. 12863, pp. 223–231. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85248-1_15
25. Hansen, N., Ostermeier, A.: Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In: Proceedings of IEEE International Conference on Evolutionary Computation, pp. 312–317 (1996)
26. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)

27. Schlüter, M., Egea, J.A., Banga, J.R.: Extended ant colony optimization for non-convex mixed integer nonlinear programming. Comput. Oper. Res. **36**(7), 2217–2229 (2009). https://www.sciencedirect.com/science/article/pii/S0305054808001524
28. Brockman, G., et al.: OpenAI gym (2016)
29. Coumans, E., Bai, Y.: Pybullet, a python module for physics simulation for games, robotics and machine learning (2016). http://pybullet.org
30. Haghighi, I., Mehdipour, N., Bartocci, E., Belta, C.: Control from signal temporal logic specifications with smooth cumulative quantitative semantics. In: 2019 IEEE 58th Conference on Decision and Control (CDC), pp. 4361–4366 (2019)
31. Mehdipour, N., Vasile, C.-I., Belta, C.: Arithmetic-geometric mean robustness for control from signal temporal logic specifications. In: 2019 American Control Conference (ACC), pp. 1690–1695 (2019)
32. Zhang, C., et al.: Tolerance of reinforcement learning controllers against deviations in cyber physical systems (2024). https://arxiv.org/abs/2406.17066
33. Saoud, A., Jagtap, P., Soudjani, S.: Temporal logic resilience for cyber-physical systems. In: 2023 62nd IEEE Conference on Decision and Control (CDC), pp. 2066–2071 (2023)
34. Fainekos, G.E., Pappas, G.J.: MTL robust testing and verification for LPV systems. In: 2009 American Control Conference, pp. 3748–3753 (2009)
35. Bartocci, E., et al.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 135–175. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_5
36. Zhang, Z., Ernst, G., Sedwards, S., Arcaini, P., Hasuo, I.: Two-layered falsification of hybrid systems guided by monte Carlo tree search. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(11), 2894–2905 (2018)
37. Zutshi, A., Deshmukh, J.V., Sankaranarayanan, S., Kapinski, J.: Multiple shooting, CEGAR-based falsification for hybrid systems. In: Proceedings of the 14th International Conference on Embedded Software, ser. EMSOFT 2014. New York, NY, USA: Association for Computing Machinery, (2014). https://doi.org/10.1145/2656045.2656061
38. Wang, J., Bu, L., Xing, S., Li, X.: PDF: path-oriented, derivative-free approach for safety falsification of nonlinear and nondeterministic CPS. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **41**(2), 238–251 (2022)
39. Dreossi, T., et al.: VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 432–442. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_25
40. Dreossi, T., Donzé, A., Seshia, S.A.: Compositional falsification of cyber-physical systems with machine learning components. J. Autom. Reason. **63**, 1031–1053 (2019)
41. Kapoor, P., Kang, E., Meira-Góes, R.: Safe planning through incremental decomposition of signal temporal logic specifications. arXiv preprint arXiv:2403.10554 (2024)
42. Lecarpentier, E., Rachelson, E.: Non-stationary Markov decision processes, a worst-case approach using model-based reinforcement learning. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc. (2019)

43. Abdullah, M.A., et al.: Wasserstein robust reinforcement learning (2019)
44. Yang, I.: A convex optimization approach to distributionally robust Markov decision processes with Wasserstein distance. IEEE Control Syst. Lett. **1**(1), 164–169 (2017)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CauMon: An Informative Online Monitor for Signal Temporal Logic

Zhenya Zhang[1(✉)], Jie An[2,3(✉)], Paolo Arcaini[3(✉)], and Ichiro Hasuo[3(✉)]

[1] Kyushu University, Fukuoka, Japan
zhang@ait.kyushu-u.ac.jp
[2] Institute of Software, Chinese Academy of Sciences, Beijing, China
anjie@iscas.ac.cn
[3] National Institute of Informatics, Tokyo, Japan
{arcaini,hasuo}@nii.ac.jp

Abstract. In this paper, we present a tool for monitoring the traces of cyber-physical systems (CPS) at runtime, with respect to *Signal Temporal Logic* (STL) specifications. Our tool is based on the recent advances of *causation monitoring*, which reports not only whether an executing trace violates the specification, but also *how relevant* the increment of the trace at each instant is to the specification violation. In this way, it can deliver more information about system evolution than classic online robust monitors. Moreover, by adapting two dynamic programming strategies, our implementation significantly improves the efficiency of causation monitoring, allowing its deployment in practice. The tool is implemented as a C++ executable and can be easily adapted to monitor CPS in different formalisms. We evaluate the efficiency of the proposed monitoring tool, and demonstrate its superiority over existing robust monitors in terms of the information it can deliver about system evolution.

Keywords: online monitoring · Signal Temporal Logic · dynamic programming

1 Introduction

Cyber-physical systems (CPS), that embed cyber technologies into physical systems, have been widely deployed in safety-critical domains, such as transportation, healthcare, power and energy. Due to their safety-critical nature, the behaviors of CPS require formal verification to guarantee their satisfaction to formal specifications that are usually expressed in temporal logics, e.g., *Signal Temporal Logic (STL)* [20]. Given an STL specification, *monitoring* (a.k.a. *runtime verification*) [2] is an effective approach for checking whether a trace of system execution satisfies the specification.

Monitoring can be achieved either *offline* or *online*. In STL monitoring, an *offline* monitor can report a real value (called *robustness*) that indicates *how*

© The Author(s) 2025

robustly an STL formula φ is satisfied or violated by a complete execution trace, based on the STL robust semantics [9,13]. By contrast, an *online* monitor targets partial execution traces at runtime, reporting the satisfaction of an STL formula φ by the partial trace so far at each time instant. Due to the lack of a complete trace, a typical *online monitor*, e.g., the *robust monitor* in [6], reports a robustness interval $[[\mathrm{R}]^{\mathsf{L}}, [\mathrm{R}]^{\mathsf{U}}]$ telling the possibly reachable values of the robustness under any suffix trace, where $[\mathrm{R}]^{\mathsf{L}}$ and $[\mathrm{R}]^{\mathsf{U}}$ are the lower and upper bounds respectively. In this way, the satisfaction of φ can be inferred from the computed robustness interval, e.g., φ is violated if $[\mathrm{R}]^{\mathsf{U}}$ is negative.

Online robust monitoring [6] suffers from the *information masking* problem [24,26,27], as visualized by the example in Fig. 1. The specification in this example requires that, during $[0, 45]$, the speed v of a car should never be over 10 for 5 time units. The top plot reports a trace v that violates the system specification. When using the classic online monitor [6] reported in the middle plot, the value of $[\mathrm{R}]^{\mathsf{U}}$ keeps on decreasing and becomes negative at $b = 10$. It then reaches the minimum value around $b = 14$ and stagnates at that value in the remaining of the monitoring. As a result, the system evolution is not faithfully reflected by the monitor. For instance, the monitor does not deliver that the system actually recovers (i.e., $v < 10$) at $b = 20$ and that the status $v > 10$ persisting for more than 5 time units happens once again from $b = 25$.

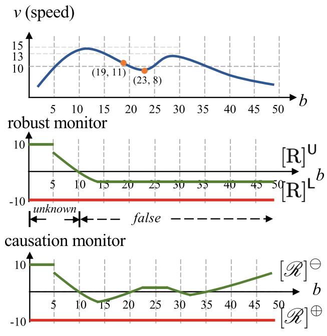


Fig. 1. An illustrative example of online robust monitoring and causation monitoring of STL formula $\varphi : \Box_{[0,45]}(\Diamond_{[0,5]} v < 10)$.

Causation monitoring [26] emerges to tackle the information masking issue of robust monitoring. As shown in the bottom plot of Fig. 1, at each instant b, an online causation monitor returns $[\mathscr{R}]^{\ominus}$ (called a *violation causation distance*) and $[\mathscr{R}]^{\oplus}$ (called a *satisfaction causation distance*), that respectively reflect how far the trace value at b is from being a *causation* to the violation and the satisfaction of the specification. For instance, $[\mathscr{R}]^{\ominus}$ at $b = 12$ is negative, which implies that the trace value at $b = 12$ is considered as a *causation* to the violation of the partial trace, because the status $v > 10$ that has been persisting for more than 5 time units continues at $b = 12$. At $b = 20$, $[\mathscr{R}]^{\ominus}$ becomes positive, which implies that the trace value at $b = 20$ is no more a causation to the violation, because v becomes less than 10 at $b = 20$. From Fig. 1, we can see that compared to robust monitoring, causation monitoring can reflect more information about system evolution, such as the recovery of the system at $b = 20$ and the recurrence of the status $v > 10$ persisting for more than 5 time units from $b = 25$.

Contributions. In this paper, we present a tool CauMon, that implements an efficient online causation monitoring algorithm of STL. Compared to the plain monitoring algorithm [26] derived directly from the definition of causation monitoring, our algorithm features the use of two dynamic programming strategies, by which we can significantly reduce redundant computation of the causation distances during monitoring of system executions. We adopt dynamic programming for two purposes: first, we record and reuse the intermediate monitoring results of the sub-formulas using several worklists for each of the sub-formulas, such that computational cost is only spent for incremental results, not for existing ones; moreover, we adapt a sliding window algorithm [18] to accelerate the computation of monitoring results in the presence of nested temporal operators.

We implement CauMon in C++, and it can be compiled to an executable that can be easily interfaced with CPS in different formalisms. We demonstrate the advantages of CauMon in informativeness, by comparing it with the existing online robust monitor in [6]. Moreover, we also evaluate the efficiency of CauMon, by comparing it with the plain causation monitor derived from the definition of causation monitoring directly, and also the online robust monitor [6]. The experimental results show that CauMon can indeed deliver more information about system evolution compared to the robust monitor; moreover, while the plain implementation of causation monitoring is not applicable to handling nested temporal operators in practice, CauMon can significantly reduce the monitoring time costs and achieve comparable efficiency with the robust monitor.

Related Work. Online monitoring is an approach that monitors system executions at runtime, and different approaches have been proposed for different temporal logics, such as LTL [4,5], MTL [14,19,25], and STL [6,7,15,16,23]. For online monitoring of STL, most of existing approaches [6,7,15,16] and tools [1,8,22,23] are based on its robust semantics and provide a quantitative value or interval to characterize the system runtime status. Consequently, these approaches suffer more or less from the issue of information masking. In [27], we proposed a reset mechanism that resets a monitor whenever it detects the recovery of specification violation. However, [27] does not propose new semantics and so it does not improve informativeness of monitors between two resets.

2 Preliminaries

2.1 Signal Temporal Logic

Let $T \in \mathbb{R}_+$ be a positive real. A *signal* (i.e., a trace of system execution) is a function $\mathbf{v}\colon [0,T] \to \mathbb{R}^d$, where T is the *time horizon*, $d \in \mathbb{N}_+$ is the dimension. In practice, each signal dimension concerns with a *signal variable* that has a certain physical meaning, e.g., `speed`, `RPM` of a car. We fix a set **Var** of variables and assume that a signal $\mathbf{v}$ is *spatially bounded* by a hyper-rectangle Ω, i.e., for any $t \in [0,T]$, $\mathbf{v}(t) \in \Omega$.

Signal temporal logic (STL) [20] can express desired properties of hybrid systems. We review the syntax and robust semantics [9,13] of STL.

Definition 1 (STL Syntax). In STL, the *atomic propositions* α and the *formulas* φ are respectively defined as follows:

$$\alpha ::\equiv f(w_1,\ldots,w_K) > 0 \qquad \varphi ::\equiv \alpha \mid \bot \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Box_I \varphi \mid \Diamond_I \varphi \mid \varphi\, \mathcal{U}_I\, \varphi$$

Here f is a K-ary function $f : \mathbb{R}^K \to \mathbb{R}$, $w_1, \dots, w_K \in \mathbf{Var}$, and I is a closed non-singular interval in $\mathbb{R}_{\geq 0}$, i.e., $I = [l, u]$, where $l, u \in \mathbb{R}$ and $l < u$. $\Box, \Diamond$ and $\mathcal{U}$ are temporal operators, which are known as *always*, *eventually* and *until*, respectively. The always operator $\Box$ and eventually operator $\Diamond$ are two special cases of the until operator $\mathcal{U}$, where $\Diamond_I \varphi \equiv \top \, \mathcal{U}_I \, \varphi$ and $\Box_I \varphi \equiv \neg \Diamond_I \neg \varphi$. Other common connectives such as $\vee, \to$ are introduced as syntactic sugar: $\varphi_1 \vee \varphi_2 \equiv \neg(\neg \varphi_1 \wedge \neg \varphi_2)$, $\varphi_1 \to \varphi_2 \equiv \neg \varphi_1 \vee \varphi_2$.

Definition 2 (STL Robust Semantics). Let $\mathbf{v}$ be a signal, φ be an STL formula and $\tau \in \mathbb{R}_+$ be an instant. The *robustness* $\mathrm{R}(\mathbf{v}, \varphi, \tau) \in \mathbb{R} \cup \{+\infty, -\infty\}$ of $\mathbf{v}$ w.r.t. φ at τ is defined by induction on the construction of formulas, as follows,

$$\begin{aligned}
&\mathrm{R}(\mathbf{v}, \alpha, \tau) \;:=\; f(\mathbf{v}(\tau)) \qquad \mathrm{R}(\mathbf{v}, \neg\varphi, \tau) \;:=\; -\mathrm{R}(\mathbf{v}, \varphi, \tau) \\
&\mathrm{R}(\mathbf{v}, \varphi_1 \wedge \varphi_2, \tau) \;:=\; \min\left(\mathrm{R}(\mathbf{v}, \varphi_1, \tau), \mathrm{R}(\mathbf{v}, \varphi_2, \tau)\right) \\
&\mathrm{R}(\mathbf{v}, \Box_I \varphi, \tau) \;:=\; \inf_{t \in \tau + I} \mathrm{R}(\mathbf{v}, \varphi, t) \qquad \mathrm{R}(\mathbf{v}, \Diamond_I \varphi, \tau) \;:=\; \sup_{t \in \tau + I} \mathrm{R}(\mathbf{v}, \varphi, t) \\
&\mathrm{R}(\mathbf{v}, \varphi_1 \, \mathcal{U}_I \, \varphi_2, \tau) \;:=\; \sup_{t \in \tau + I} \min\left(\mathrm{R}(\mathbf{v}, \varphi_2, t), \inf_{t' \in [\tau, t)} \mathrm{R}(\mathbf{v}, \varphi_1, t')\right)
\end{aligned}$$

where $\tau + [l, u]$ denotes the shifted interval $[l + \tau, u + \tau]$.

The Boolean semantics of STL, i.e., whether $(\mathbf{v}, \tau) \models \varphi$ or not, can be inferred from the quantitative robust semantics in Definition 2, namely, if $\mathrm{R}(\mathbf{v}, \varphi, \tau) > 0$, it implies $(\mathbf{v}, \tau) \models \varphi$; and if $\mathrm{R}(\mathbf{v}, \varphi, \tau) < 0$, it implies $(\mathbf{v}, \tau) \not\models \varphi$.

2.2 Online Robust Monitoring of STL

Online monitoring concerns the satisfaction of a *partial signal* $\mathbf{v}_{0:b} : [0, b] \to \mathbb{R}^d$ w.r.t. an STL formula φ. We define a *completion* of $\mathbf{v}_{0:b}$ as a signal $\mathbf{v} : [0, T] \to \mathbb{R}^d$ ($b \leq T$) such that $\forall t \in [0, b], \mathbf{v}(t) = \mathbf{v}_{0:b}(t)$. A completion $\mathbf{v}$ can be written as the concatenation of $\mathbf{v}_{0:b}$ with a *suffix signal* $\mathbf{v}_{b:T}$, i.e., $\mathbf{v} = \mathbf{v}_{0:b} \cdot \mathbf{v}_{b:T}$.

Definition 3 (Online Robust Monitor [6]). Let $\mathbf{v}_{0:b}$ be a partial signal, and let φ be an STL formula. We denote by $\mathtt{R}^{\alpha}_{\max}$ and $\mathtt{R}^{\alpha}_{\min}$ the possible *maximum* and *minimum bounds* of the robustness $\mathrm{R}(\mathbf{v}, \alpha, \tau)$[1]. Then, an *online robust monitor* returns a sub-interval $[\mathrm{R}](\mathbf{v}_{0:b}, \varphi, \tau) \subseteq [\mathtt{R}^{\alpha}_{\min}, \mathtt{R}^{\alpha}_{\max}]$ at instant b, which is defined as follows, by induction on the construction of formulas.

$$[\mathrm{R}](\mathbf{v}_{0:b}, \alpha, \tau) \;:=\; \begin{cases} \left[f\left(\mathbf{v}_{0:b}(\tau)\right), f\left(\mathbf{v}_{0:b}(\tau)\right)\right] & \text{if } \tau \in [0, b] \\ \left[\mathtt{R}^{\alpha}_{\min}, \mathtt{R}^{\alpha}_{\max}\right] & \text{otherwise} \end{cases}$$

$$[\mathrm{R}](\mathbf{v}_{0:b}, \neg\varphi, \tau) \;:=\; -[\mathrm{R}](\mathbf{v}_{0:b}, \varphi, \tau)$$

$$[\mathrm{R}](\mathbf{v}_{0:b}, \varphi_1 \wedge \varphi_2, \tau) \;:=\; \min\Big([\mathrm{R}](\mathbf{v}_{0:b}, \varphi_1, \tau), [\mathrm{R}](\mathbf{v}_{0:b}, \varphi_2, \tau)\Big)$$

$$[\mathrm{R}](\mathbf{v}_{0:b}, \Box_I \varphi, \tau) \;:=\; \inf_{t \in \tau + I} \Big([\mathrm{R}](\mathbf{v}_{0:b}, \varphi, t)\Big)$$

$$[\mathrm{R}](\mathbf{v}_{0:b}, \varphi_1 \, \mathcal{U}_I \, \varphi_2, \tau) \;:=\; \sup_{t \in \tau + I} \min\left([\mathrm{R}](\mathbf{v}_{0:b}, \varphi_2, t), \inf_{t' \in [\tau, t)} [\mathrm{R}](\mathbf{v}_{0:b}, \varphi_1, t')\right)$$

[1] $\mathrm{R}(\mathbf{v}, \alpha, \tau)$ is bounded because of the bound Ω of $\mathbf{v}$. In practice, if Ω is unknown, we just need to set $\mathtt{R}^{\alpha}_{\max}$ and $\mathtt{R}^{\alpha}_{\min}$ to be ∞ and $-\infty$ respectively.

Here, f is defined as in Definition 1, and the arithmetic rules over intervals $I = [l, u]$ are defined as follows: $-I := [-u, -l]$ and $\min(I_1, I_2) := [\min(l_1, l_2), \min(u_1, u_2)]$.

We denote by $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau)$ and $[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, \tau)$ the upper and lower bounds of $[\mathrm{R}](\mathbf{v}_{0:b}, \varphi, \tau)$ respectively. Intuitively, this interval $[\mathrm{R}](\mathbf{v}_{0:b}, \varphi, \tau)$ indicates the set of robustness values possibly reached by the completion of $\mathbf{v}_{0:b}$, under any suffix signal $\mathbf{v}_{b:T}$. This interval can be used to derive a 3-valued verdict for a given $\mathbf{v}_{0:b}$, that signifies the satisfaction of $\mathbf{v}_{0:b}$ w.r.t. the specification φ: if $[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, \tau) > 0$, it implies `true`, i.e., $\mathbf{v}_{0:b}$ satisfies φ; if $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau) < 0$, it implies `false`, i.e., $\mathbf{v}_{0:b}$ violates φ; otherwise, it returns `unknown`.

3 Overview of Causation Monitoring

As mentioned in §1, the information masking issue of online robust monitors has been identified as a problem in [24,26,27]. The problem arises from the *monotonicity* of online robust monitors, i.e., during evolution of the signal $\mathbf{v}_{0:b}$, $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau)$ monotonically decreases and $[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, \tau)$ monotonically increases. The formal statement of this problem can be found in [26].

Online Causation Monitoring. *Causation monitoring* is proposed in [26] as a solution to the problem. Specifically, instead of monitoring robustness that indicates whether a partial trace violates the specification, it monitors whether the increment of the trace at each instant is the *causation* to the violation of the specification. Here, the definition of causation follows the online trace diagnostics [3,27] that returns a (violation or satisfaction) *epoch*, which is a set of signal segments that *sufficiently* triggers the violation or satisfaction of the partial signal. Intuitively, an epoch can be considered as an explanation of a violation or satisfaction; the formal definition of epoch can be found in [3,27]. Having the trace diagnostic result at each instant, causation monitoring aims to report such a verdict: if the current instant b of $\mathbf{v}_{0:b}$ is included in the violation epoch, it is considered as a *violation causation*; if b is included in the satisfaction epoch, it is considered as a *satisfaction causation*; otherwise, it is *irrelevant*.

The causation monitor proposed in [26] achieves this goal as follows: at each instant, it computes two quantities $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau)$ and $[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau)$, that respectively indicate the distances of the current instant b from being a violation causation and a satisfaction causation. The formal definition of causation distances $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau)$ and $[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau)$ are presented in Definition 4.

Definition 4 (Online Causation Monitor [26]). Let $\mathbf{v}_{0:b}$ be a partial signal and φ be an STL formula. At an instant b, an online causation monitor returns a *violation causation distance* $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau)$ and a *satisfaction causation distance* $[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau)$, as defined in Table 1.

The causation verdict, regarding whether b is a causation or not, can be inferred by the results of the online causation monitor in Definition 4, as follows:

Table 1. The definitions of violation and satisfaction causation distances

$$[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \alpha, \tau) := \begin{cases} f(\mathbf{v}_{0:b}(\tau)) & \text{if } b = \tau \\ \mathtt{R}^{\alpha}_{\max} & \text{otherwise} \end{cases} \qquad [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \neg\varphi, \tau) := -[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau)$$

$$[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \wedge \varphi_2, \tau) := \min\left([\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1, \tau), [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_2, \tau)\right)$$

$$[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \Box_I \varphi, \tau) := \inf_{t \in \tau + I}\left([\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, t)\right)$$

$$[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \,\mathcal{U}_I\, \varphi_2, \tau) := \inf_{t \in \tau + I}\left(\max\left(\begin{array}{l} \min\left(\begin{array}{l} \inf_{t' \in [\tau, t)} [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1, t') \\ [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_2, t) \end{array}\right) \\ [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi_1 \,\mathcal{U}_I\, \varphi_2, \tau) \end{array}\right)\right)$$

$$[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \alpha, \tau) := \begin{cases} f(\mathbf{v}_{0:b}(\tau)) & \text{if } b = \tau \\ \mathtt{R}^{\alpha}_{\min} & \text{otherwise} \end{cases} \qquad [\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \neg\varphi, \tau) := -[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau)$$

$$[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_1 \wedge \varphi_2, \tau) := \max\left(\begin{array}{l} \min\left([\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_1, \tau), [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi_2, \tau)\right) \\ \min\left([\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi_1, \tau), [\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_2, \tau)\right) \end{array}\right)$$

$$[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \Box_I \varphi, \tau) := \sup_{t \in \tau + I}\left(\min\left([\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, t), [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \Box_I \varphi, \tau)\right)\right)$$

$$[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_1 \,\mathcal{U}_I\, \varphi_2, \tau) := \sup_{t \in \tau + I}\left(\max\left(\begin{array}{l} \min\left(\begin{array}{l} \sup_{t' \in [\tau, t)} [\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_1, t') \\ \inf_{t' \in [\tau, t)} [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi_1, t') \\ [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi_2, t) \end{array}\right) \\ \min\left(\begin{array}{l} \inf_{t' \in [\tau, t)} [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi_1, t') \\ [\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_2, t) \end{array}\right) \end{array}\right)\right)$$

- if $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau) < 0$, then b is a violation causation;
- if $[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau) > 0$, then b is a satisfaction causation;
- otherwise, i.e., $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau) > 0$ and $[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau) < 0$, b is irrelevant.

Below, we use an example to illustrate how online causation monitor works.

Example 1. Consider the example in Fig. 1. As indicated by the robust monitor, the specification is violated by the signal after $b = 10$. By online trace diagnostics (see [27]), at $b = 19$, we can obtain a violation epoch $\{\langle v < 10, t\rangle \mid t \in [5, 19]\}$, which implies that the violation so far is caused by the signal values v during $[5, 19]$. Since $b = 19$ is included in this epoch, b is then considered as a violation causation. On the other hand, we can also compute the violation causation distance $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, 0) = -1 < 0$ by Definition 4, which also indicates that b is a violation causation.

Similarly, at $b = 23$, we obtain an epoch $\{\langle v < 10, t\rangle \mid t \in [5, 20]\}$, in which $b = 23$ is not included, so $b = 23$ is irrelevant. This is also shown by computing the causation distances $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, 0) = 2 > 0$ and $[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, 0) = \mathtt{R}^{\alpha}_{\min} < 0$.

Note that the result of causation monitoring is not monotonic, e.g., while $b = 19$ is considered as a violation causation, $b = 23$ is not. This feature is clearly shown by the visualized result of causation monitoring in Fig. 1.

Relationship with Robust Monitors. As indicated by Fig. 1 and Example 1, the causation monitor is not monotonic, and thus it delivers more information about system evolution. We refer to [26] for a more detailed explanation. Below, Lemma 1 states that the online causation monitor in Definition 4 refines the online robust monitor in Definition 3, in the sense that the monitoring results of robust monitors can be inferred from that of causation monitors. In other words, the information delivered by causation monitors is a superset of that can be delivered by classic robust monitors.

Lemma 1. The causation monitor in Definition 4 refines the classic online robust monitor in Definition 3, in the sense that the monitoring results of the robust monitor can be reconstructed from the results of the causation monitor, as follows:

$$[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau) = \inf_{t \in [0,b]} [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:t}, \varphi, \tau), \ [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, \tau) = \sup_{t \in [0,b]} [\mathscr{R}]^{\oplus}(\mathbf{v}_{0:t}, \varphi, \tau)$$

4 Efficient Causation Monitoring

In [26], a straightforward way of synthesizing an online causation monitor has been provided that follows Definition 4. However, the synthesized monitor may not be sufficiently efficient to be deployed in practice, due to the high computational complexity when handling nested temporal operators.

Example 2. Consider the specification $\varphi \equiv \Box_{[0,45]}(\Diamond_{[0,5]} v < 10)$ in Fig. 1. For convenience, we name the sub-formulas of φ as follows: $\varphi' \equiv \Diamond_{[0,5]}(v < 10), \alpha \equiv (v < 10)$. Consider the computation of $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, 0)$ that contains nested temporal operators. According to Definition 4, we need to compute as follows:

$$\begin{aligned}
&[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, 0) = \inf\nolimits_{t \in [0,45]} [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi', t) \\
&= \inf\nolimits_{t \in [0,45]} \left(\inf\nolimits_{t' \in [t,t+5]} \left(\max \left([\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \alpha, t'), [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \alpha, t') \right) \right) \right)
\end{aligned}$$

During this computation, for a fixed t', $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \alpha, t')$ and $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \alpha, t')$ are repeatedly computed as long as it holds that $t' \in [t, t+5]$ for any $t \in [0, 45]$. This results in numerous redundant computations, which can significantly diminish the efficiency of causation monitoring.

We introduce two dynamic programming strategies, i.e., *intermediate result recording* and *sliding window*, for accelerating causation monitoring.

4.1 Intermediate Result Recording

Our efficient causation monitoring algorithm is presented in Algorithm 1. The basic idea of the algorithm is to record the intermediate monitoring results, by maintaining several worklists for each of the sub-formulas of an STL formula φ, so as to avoid redundant computations.

Algorithm 1 Efficient online causation monitoring

```
Require: a partial signal $\mathbf{v}_{0:b}$, an STL formula $\varphi$
 1: for $\psi \in \mathsf{SF}(\varphi)$ do
 2:     for $t \in \mathit{Eva}(\varphi, 0)[\psi]$ do
 3:         $\mathsf{Cau}^{\ominus}[\psi](t) \leftarrow \mathtt{R}^{\alpha}_{\max}$, $\mathsf{Cau}^{\oplus}[\psi](t) \leftarrow \mathtt{R}^{\alpha}_{\min}$
 4:         $\mathsf{Rob}^{\mathsf{U}}[\psi](t) \leftarrow \mathtt{R}^{\alpha}_{\max}$, $\mathsf{Rob}^{\mathsf{L}}[\psi](t) \leftarrow \mathtt{R}^{\alpha}_{\min}$
 5: for $b \in [0, T]$ do
 6:     UpdateCau($\mathbf{v}_{0:b}, \varphi, 0$)                          ▷ monitoring at runtime
 7: function UpdateCau($\mathbf{v}_{0:b}, \psi, \tau$)
 8:     switch $\psi$ do
 9:         case $\alpha$                                               ▷ atomic propositions
10:             if $b \in \mathit{Eva}(\alpha)$ then
11:                 $\mathsf{Cau}^{\ominus}[\psi](t) \leftarrow \begin{cases} f(\mathbf{v}_{0:b}(b)) & \text{if } t = b \\ +\infty & \text{otherwise} \end{cases}$
12:                 $\mathsf{Cau}^{\oplus}[\psi](t) \leftarrow \begin{cases} f(\mathbf{v}_{0:b}(b)) & \text{if } t = b \\ -\infty & \text{otherwise} \end{cases}$
13:         case $\neg\varphi$                                          ▷ negations
14:             UpdateCau($\mathbf{v}_{0:b}, \varphi, \tau$)                ▷ recursive call
15:             $\mathsf{Cau}^{\ominus}[\psi] \leftarrow -\mathsf{Cau}^{\oplus}[\varphi]$
16:             $\mathsf{Cau}^{\oplus}[\psi] \leftarrow -\mathsf{Cau}^{\ominus}[\varphi]$
17:         case $\varphi_1 \wedge \varphi_2$                            ▷ conjunctions
18:             UpdateCau($\mathbf{v}_{0:b}, \varphi_1, \tau$); UpdateCau($\mathbf{v}_{0:b}, \varphi_2, \tau$)   ▷ recursive call
19:             UpdateRob($\mathbf{v}_{0:b}, \varphi_1, \tau$); UpdateRob($\mathbf{v}_{0:b}, \varphi_2, \tau$)   ▷ see [6]
20:             $\mathsf{Cau}^{\ominus}[\psi] \leftarrow \min(\mathsf{Cau}^{\ominus}[\varphi_1], \mathsf{Cau}^{\ominus}[\varphi_2])$
21:             $\mathsf{Cau}^{\oplus}[\psi] \leftarrow \max\big(\min(\mathsf{Cau}^{\oplus}[\varphi_1], \mathsf{Rob}^{\mathsf{L}}[\varphi_2]), \min(\mathsf{Rob}^{\mathsf{L}}[\varphi_1], \mathsf{Cau}^{\oplus}[\varphi_2])\big)$
22:         case $\Box_I \varphi$                                       ▷ always operators
23:             UpdateCau($\mathbf{v}_{0:b}, \varphi, \tau$)                ▷ recursive call
24:             UpdateRob($\mathbf{v}_{0:b}, \Box_I \varphi, \tau$)          ▷ see [6]
25:             $\mathsf{Cau}^{\ominus}[\psi] \leftarrow$ SlideMin($\mathsf{Cau}^{\ominus}[\varphi], \mathsf{Trans}(I)$)   ▷ call Algorithm 2
26:             $\mathsf{Cau}^{\oplus}[\psi] \leftarrow \min\big(\mathsf{Rob}^{\mathsf{L}}[\varphi], -$SlideMin$(-\mathsf{Cau}^{\oplus}[\varphi], \mathsf{Trans}(I))\big)$
```

Sub-formulas and Evaluation periods. Given an STL formula φ, the sub-formula set $\mathsf{SF}(\varphi)$ of φ is defined as follows (see an example in Example 2):

$$\mathsf{SF}(\alpha) := \{\alpha\} \qquad \mathsf{SF}(\neg\varphi) := \{\neg\varphi\} \cup \mathsf{SF}(\varphi)$$
$$\mathsf{SF}(\varphi_1 \wedge \varphi_2) := \{\varphi_1 \wedge \varphi_2\} \cup \mathsf{SF}(\varphi_1) \cup \mathsf{SF}(\varphi_2) \qquad \mathsf{SF}(\Box_I \varphi) := \{\Box_I \varphi\} \cup \mathsf{SF}(\varphi)$$

At the beginning of Algorithm 1, for each sub-formula $\psi \in \mathsf{SF}(\varphi)$, we initialize four worklists, including $\mathsf{Cau}^{\ominus}[\psi]$ and $\mathsf{Cau}^{\oplus}[\psi]$ that record the violation and satisfaction causation distances, $\mathsf{Rob}^{\mathsf{U}}[\psi]$ and $\mathsf{Rob}^{\mathsf{L}}[\psi]$ that record the upper and lower robustness bounds.

Each of these four lists is defined over the *evaluation period* $\mathit{Eva}(\psi)$ of each ψ, that is, intuitively, the time interval that includes all the instants t such that $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \psi, t)$ is needed for the computation of $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, 0)$ (see Definition 4, and it also holds for satisfaction distances). Formally, given $\psi \in \mathsf{SF}(\varphi)$, the evaluation period is computed as follows. First, $Eva[\varphi, 0]$, a set that includes the evaluation periods of all the sub-formulas $\psi' \in \mathsf{SF}(\varphi)$, is computed recursively as follows:

$$\begin{aligned}&Eva[\alpha, t] := \{\langle \alpha, t\rangle\} \qquad Eva[\neg\varphi, t] := \{\langle \neg\varphi, t\rangle\} \cup Eva[\varphi, t]\\&Eva[\varphi_1 \wedge \varphi_2, t] := \{\langle \varphi_1 \wedge \varphi_2, t\rangle\} \cup Eva[\varphi_1, t] \cup Eva[\varphi_2, t]\\&Eva[\Box_I \varphi, t] := \{\langle \Box_I \varphi, t\rangle\} \cup \textstyle\bigcup_{t' \in t+I} Eva[\varphi, t']\end{aligned}$$

Then, the evaluation period $Eva[\varphi, 0](\psi)$ (we denote as $Eva(\psi)$ for simplicity) of ψ is defined as $Eva(\psi) = \{t \mid \langle \psi, t\rangle \in Eva[\varphi, 0]\}$.

Example 3. Consider the sub-formulas φ, φ' and α of the formula φ in Example 2. The evaluation periods of these sub-formulas can be computed as follows.

- First, $Eva[\varphi, 0] = \{\langle \varphi, 0\rangle\} \cup \bigcup_{t' \in [0,45]} \{\langle \varphi', t'\rangle\} \cup \bigcup_{t'' \in [0,50]} \{\langle \varphi', t''\rangle\}$;
- Then, we can obtain the evaluation periods for φ, φ' and α, respectively as follows: $Eva(\varphi) = 0, Eva(\varphi') = [0, 45], Eva(\alpha) = [0, 50]$.

Monitoring Algorithm. During the growth of partial signal $\mathbf{v}_{0:b}$, Algorithm 1 monitors $\mathbf{v}_{0:b}$ by calling the function UPDATECAU at each instant, which updates the worklists $\mathsf{Cau}^{\ominus}[\psi]$ and $\mathsf{Cau}^{\oplus}[\psi]$, such that $\mathsf{Cau}^{\ominus}[\psi](t)$ equals to $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \psi, t)$ and $\mathsf{Cau}^{\oplus}[\psi](t)$ equals to $[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \psi, t)$, as defined in Definition 4. Unlike the plain monitoring algorithm in Example 2, when updating $\mathsf{Cau}^{\ominus}[\psi]$ and $\mathsf{Cau}^{\oplus}[\psi]$, Algorithm 1 relies on the worklists of the sub-formulas of ψ that are already available rather than computing the causation distances of the sub-formulas from scratch, thereby saving monitoring time significantly. We illustrate this process in Example 4.

As shown in Algorithm 1, UPDATECAU is defined recursively based on the structure of an STL formula. In Algorithm 1, we only show a part of the operators; other operators can be derived by the STL syntax (Definition 1) and the presented operators.

- The updates for α and $\neg\varphi$ exactly follow Definition 4;
- The update for $\varphi_1 \wedge \varphi_2$ requires not only the worklists of causation distances of sub-formulas, but also the worklists of robustness bounds of sub-formulas (according to Definition 4), so it calls the auxiliary function UPDATEROB (see Algorithm 2 in [6]) to update the worklists $\mathsf{Rob}^{\mathsf{L}}[\varphi_1]$ and $\mathsf{Rob}^{\mathsf{L}}[\varphi_2]$ of robustness bounds of sub-formulas. The function UPDATEROB was originally introduced in [6]. It updates the worklists of robustness bounds in a similar way to what UPDATECAU does for causation distances, i.e., when updating the worklists of robustness bounds for a formula ψ, it also relies on the worklists of robustness bounds for its sub-formulas, rather than computing from scratch.
- The update for $\Box_I \varphi$ requires to compute the minimum of $\mathsf{Cau}^{\ominus}[\varphi]$ over the time window $t + I$, for each $t \in Eva(\Box_I \varphi)$. To efficiently update the list, we adapt a sliding window algorithm [18] (elaborated on in §4.2) that, given a list

Algorithm 2 Sliding window algorithm

Require: a list $A = \{a_1, \ldots, a_N\}$, a window $\omega = [l, u]$ $(l, u \in \mathbb{N})$
Ensure: a list $\mathsf{result} = \{\min_{j\in[i+l,i+u]} a_j \mid i \in \{1, \ldots, N-u\}\}$

```
 1: function SlideMin($A, \omega$)
 2:     $Q \leftarrow$ empty double-ended queue
 3:     $\mathsf{result} \leftarrow \emptyset$                          ▷ initialize the list of results
 4:     PushBack($Q, l+1$)
 5:     for $i \in \{l+2, \ldots, N\}$ do
 6:         if $i \geq u+1$ then                            ▷ should give outputs
 7:             $\mathsf{result} \leftarrow \mathsf{result} \cup \{a_{\mathrm{Front}(Q)}\}$      ▷ record result
 8:         if $a_i < a_{i-1}$ then              ▷ the back is not possible in result
 9:             PopBack($Q$)                                  ▷ remove back
10:             while $a_i < a_{\mathrm{Back}(Q)}$ do              ▷ recursive check
11:                 PopBack($Q$)                              ▷ remove back
12:         PushBack($Q, i$)                          ▷ $a_i$ possibly in result
13:         if $i > \mathrm{Front}(Q) + u - l$ then      ▷ front is no more in window
14:             PopFront($Q$)                                ▷ remove front
15:     return result
```

$\{a_1, \ldots, a_N\}$ and an index window $[l, u]$ $(l, u \in \mathbb{N})$, computes the minimum $\min_{j\in[i+l,i+u]} a_j$ over the window $[i+l, i+u]$ for each $i \in \{1, \ldots, N-u\}$. In Algorithm 1, $\mathsf{Trans}(I)$ transforms a time interval I to the index representation of a window, simply by considering the sampling frequency[2].

4.2 Sliding Window Algorithm

The sliding window algorithm [18] is given in Algorithm 2, which computes the local minimum $\min_{j\in[i+l,i+u]} a_j$ over the span $[i+l, i+u]$, for each $i \in \{1, \ldots, N-u\}$. This is yet another dynamic programming strategy, by using a double-ended queue Q (Line 2) to record the comparisons that have been performed between the elements in Q, and thus eliminate redundant comparisons.

Algorithm 2 takes as input a list $\{a_1, \ldots, a_N\}$ and a window $[l, u]$. Initially, the window is placed to contain the elements $a_{1+l}, \ldots, a_{1+u}$, and the index $l+1$ is pushed to the back of Q (Line 4). Then it enters a loop to traverse the list (Line 5). Inside the loop, first, it checks the condition whether a result should be reported, i.e., the elements in the initial window have been traversed (Line 6). From which that on, it reports the list element with the index at the front of Q at each loop (Line 7). Then, it recursively removes the indexes $a_{\mathrm{BACK(Q)}}$ at the back of Q, if the element a_i of the current index i is greater than the $a_{\mathrm{BACK(Q)}}$ (Line 8-11), and pushes the current index i to the back of Q (Line 12). If the

[2] In practice, the continuous time domain of signals (see §2.1) needs to be discretized, by sampling the signal with a certain frequency. In this way, a signal can be represented as a list, which is the format required in Algorithm 2.

index at the front of Q is already out of the scope of the window, then it is also removed (Line 14).

Note that, the index of local minimum over the window is always stored at the front of Q, and that is why it is returned at each loop in Line 7 of Algorithm 2. In this way, the comparisons that have been performed between list elements are reflected by the state of Q, thus reducing redundancies significantly.

Example 4. Consider the specification in Example 2. According to Algorithm 1, our computation of $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, 0)$ at $b = 19$ relies on updating the worklists for the sub-formulas $\mathsf{SF}(\varphi)$ of φ. As shown in Table 2, we need to maintain five worklists for the sub-formulas of φ. Each worklist is defined over the evaluation period of the sub-formula, as computed in Example 3.

Due to recursive call of UpdateCau and UpdateRob, our algorithm first updates the worklist $\mathsf{Cau}^{\ominus}[\alpha]$ and $\mathsf{Rob}^{\mathsf{U}}[\alpha]$ for α, as shown by the corresponding rows in Table 2. Then, $\mathsf{Rob}^{\mathsf{U}}[\varphi']$ is updated by taking the local maximum over each window $[0, 5]$ (by UpdateRob), and $\mathsf{Cau}^{\ominus}[\varphi']$ is updated based on $\mathsf{Cau}^{\ominus}[\alpha]$ and $\mathsf{Rob}^{\mathsf{U}}[\varphi']$ (by Algorithm 1). Finally, $\mathsf{Cau}^{\ominus}[\varphi]$ can be updated based on $\mathsf{Cau}^{\ominus}[\varphi']$.

Table 2. The worklists for computing $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, 0)$ at $b = 19$

b (time)	0	⋯	5	⋯	10	⋯	14	15	⋯	19	20	⋯	45	⋯	50
$\mathsf{Cau}^{\ominus}[\alpha]$	$\mathrm{R}^{\alpha}_{\max}$	⋯	$\mathrm{R}^{\alpha}_{\max}$	⋯	$\mathrm{R}^{\alpha}_{\max}$	⋯	$\mathrm{R}^{\alpha}_{\max}$	$\mathrm{R}^{\alpha}_{\max}$	⋯	-1	$\mathrm{R}^{\alpha}_{\max}$	⋯	$\mathrm{R}^{\alpha}_{\max}$	⋯	$\mathrm{R}^{\alpha}_{\max}$
$\mathsf{Rob}^{\mathsf{U}}[\alpha]$	10	⋯	0	⋯	-4	⋯	-3.5	-3	⋯	-1	$\mathrm{R}^{\alpha}_{\max}$	⋯	$\mathrm{R}^{\alpha}_{\max}$	⋯	$\mathrm{R}^{\alpha}_{\max}$
$\mathsf{Rob}^{\mathsf{U}}[\varphi']$	10	⋯	0	⋯	-3	⋯	-1	$\mathrm{R}^{\alpha}_{\max}$	⋯	$\mathrm{R}^{\alpha}_{\max}$	$\mathrm{R}^{\alpha}_{\max}$	⋯	$\mathrm{R}^{\alpha}_{\max}$		
$\mathsf{Cau}^{\ominus}[\varphi']$	$\mathrm{R}^{\alpha}_{\max}$	⋯	$\mathrm{R}^{\alpha}_{\max}$		$\mathrm{R}^{\alpha}_{\max}$		-1	$\mathrm{R}^{\alpha}_{\max}$		$\mathrm{R}^{\alpha}_{\max}$	$\mathrm{R}^{\alpha}_{\max}$	⋯	$\mathrm{R}^{\alpha}_{\max}$		
$\mathsf{Cau}^{\ominus}[\varphi]$	-1														

Compare our algorithm with the naive one in Example 2. In our algorithm, the computations of $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \alpha, t)$ and $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \alpha, t)$ for any specific t both happen only once; then, they are recorded in the worklists and when they are used to update the worklists of other formulas, they can be directly read from the worklist, which does not take new computation costs. Therefore, our algorithm avoids the repeated computation of $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \alpha, t)$ and $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \alpha, t)$, as shown in the monitoring process in Example 2.

5 Demonstration of CauMon

We implemented CauMon in `C++`, which can be easily compiled to interface with CPS implemented in any formalism. In this section, we showcase the usage of CauMon, by compiling it to be a MATLAB API, based on the MEX functions of MATLAB. A code snippet for monitoring φ in Fig. 1 is shown as follows.

```
signal = 'speed';
spec = 'alw_[0,45](ev_[0,5](speed[t]<10))';
tau = 0;
while ~end()
    trace = get_trace();
    [vio_d, sat_d] = cau_mon(signal, trace, spec, tau);
end
```

The function `cau_mon` serves as the interface to call our causation monitor, which requires four arguments, namely, a trace, a list of signal names, an STL specification and a non-negative τ. Their formats are required as follows:

- `trace` is a high-dimensional array. Its first row is an array of time stamps; from second row on, each row denotes a signal concerned with the STL;
- `signal` is a string that denotes a list of signals that correspond to each row of `trace`. Multiple signals can be separated by commas.
- `spec` is a string that denotes an STL formula. The syntax of `spec` follows the standard in Breach [8].
- `tau` is a non-negative real, as τ defined in Definition 4.

During the monitoring of system execution, the program iteratively calls `cau_mon` with an updated `trace`, which is obtained by a function `get_trace`. The function `get_trace` requires an interface with the system being monitored. Typically, such an interface is provided by a CPS simulator; for example, in the case of Simulink models, one can use the function `sim` to obtain the output of Simlink models. At each instant, `cau_mon` can return a violation causation distance `vio_d` and a satisfaction causation distance `sat_d`, exactly as defined in Definition 4.

6 Experimental Evaluation

We introduce the experimental evaluation of our tool CauMon. For the purpose of comparison, we also integrated two baseline monitors, namely, RobM (the online robust monitor from [6]) and PCauM (the plain implementation of online causation monitor from Definition 4) into our tool, as an option that can be specified by users. Our tool is publicly available in our Github repository[3].

6.1 Experiment Setting

Benchmarks. To evaluate the efficiency of our proposed monitoring algorithm, we collect traces from four MATLAB Simulink models that are commonly-used in the CPS community [10–12, 21].

Automatic Transmission (AT) implements a transmission controller of an automotive system. It has been widely-used recently [10–12] as a benchmark of CPS testing and monitoring. It contains 64 blocks in total, including a *stateflow* chart

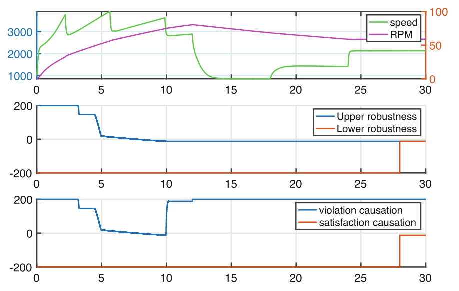
[3] https://github.com/choshina/EfficientCausationMonitor.

that represents the transmission control logic. The outputs of AT, including `speed` and `RPM`, reflect the state of the automotive system. The specifications that AT are expected to hold are listed as follows:

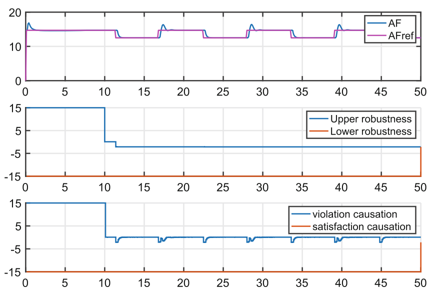
- $\varphi_1^{\mathsf{AT}} \equiv \square_{[0,30]}(\texttt{speed} < 110)$: `speed` should be always low;
- $\varphi_2^{\mathsf{AT}} \equiv \square_{[0,29]}(\texttt{speed} > 70 \rightarrow \Diamond_{[0,1]}(\texttt{speed} > 80))$: there should be a drastic `speed` change from 70 to 80;
- $\varphi_3^{\mathsf{AT}} \equiv \square_{[0,27]}(\texttt{speed} > 50 \rightarrow \Diamond_{[1,3]}(\texttt{RPM} < 3000))$: whenever `speed` is higher than 50, `RPM` should be below 3000 in 3 s;
- $\varphi_4^{\mathsf{AT}} \equiv \square_{[0,29]}(\texttt{speed} < 100) \vee \square_{[29,30]}(\texttt{speed} > 65)$: there should not be a drastic speed change at the end of the simulation;

Abstract Fuel Control (AFC) is a powertrain control system released by Toyota [17] and has been widely used as a benchmark in CPS community [10–12, 21]. The system takes external inputs including engine speed and pedal angle, and adjusts the *air-to-fuel ratio* to ensure the performance of the powertrain system. The output of AFC includes the *air-to-fuel ratio* `AF` and a reference value `AFref`. The specifications of AFC are listed as follows:

- $\varphi_1^{\mathsf{AFC}} \equiv \square_{[10,50]}(|\texttt{AF} - \texttt{AFref}| < 0.1)$: the deviation of `AF` from `AFref` should always be small;
- $\varphi_2^{\mathsf{AFC}} \equiv \square_{[10,48.5]}\Diamond_{[0,1.5]}(|\texttt{AF} - \texttt{AFref}| < 0.08)$: a large deviation should not last for too long;



(a) A trace with specification φ_3^{AT}



(b) A trace with specification φ_1^{AFC}

Fig. 2. Comparison between CauMon and RobM, in terms of the information provided by different monitors. In each of the sub-figures, the top plot is the signals, the middle plot is the result of online robust monitors [6], and the bottom plot is the result of online causation monitors.

Neural Network Controller (NN). This is a magnetic levitation system that has been used as a benchmark in [10, 12, 21, 29]. It takes one input signal, $Ref \in [1, 3]$, which is the reference for the position Pos of a magnet suspended above an electromagnet. The specification of NN is a complicated one:

- $\varphi_1^{\mathsf{NN}} \equiv \square_{[0,18]}(\neg \texttt{close} \rightarrow \texttt{reach})$, where $\texttt{close} \equiv |Pos - Ref| \leq \rho + a \cdot |Ref|$, and $\texttt{reach} \equiv \diamond_{[0,2]}(\square_{[0,1]}(\texttt{close}))$: the position should approach the reference position in some seconds when they are far. Here, $a = 0.04$ and $\rho = 0.004$.

Free Floating Robot (FFR) models a robot moving in a 2D space [28,30]. It takes as input the four boosters of the robot, and outputs four signals that are the position in terms of coordinate values x, y. The specification of FFR is as follows:

- $\varphi_1^{\mathsf{FFR}} \equiv \neg(\diamond_{[0,5]}(\square_{[0,2]}(x \in [1.5, 1.7] \wedge y \in [1.5, 1.7])))$: it requires the robot not to stay in an area for at least 2 s.

Experiment Design. For each specification, we first generate 10 signals by running the Simulink models with random inputs, and then for each signal, we apply the three monitors and compare the total time they spent in monitoring the signal. To handle the fluctuation of monitoring time due to the environmental noises, we repeat each monitoring process for five times and report the average monitoring time as the result. While some monitors may be not efficient and not terminated within a reasonable time budget, we set 5000 s as a timeout.

Our experiments are conducted on Amazon EC2 c4.2xlarge instances (2.9 GHz Intel Xeon E5-2666 v3, 15 GB RAM).

6.2 Evaluation

Efficacy of CauMon. To demonstrate the supriority of CauMon compared to RobM in terms of informativeness, we show two plots in Fig. 2, which depict the signals produced by Simulink models and the monitoring results of CauMon and RobM. Due to the page limit, examples of other specifications are presented in our Github repository. We can observe that, while the upper robustness curves in these three plots provided by RobM are always monotonic, the violation causation distances provided by CauMon are not monotonic, thus they can deliver more information about system evolution. For instance, in Fig. 2b, the spikes shown in the monitoring result of CauMon reflect that the deviation between `AF` and `AFref` is greater than the threshold 0.1 in φ_1^{AFC} for more than once. However, this information can not be provided by the monotonic curve of the upper robustness from the monitoring result given by RobM.

Efficiency of CauMon. The experimental results are presented in Table 3. Each sub-table reports the results of monitoring 10 signals for the corresponding specification. The first three columns of each sub-table report the total monitoring time of the three monitors, and the last two columns report the comparison between the proposed CauMon and the two baseline monitors RobM and CauMon, computed by $\Delta A = {}^{(\mathsf{CauMon} - A)}/_{A}$, where A is either PCauM or RobM.

First, by the comparison between PCauM and CauMon, we observe that in our experiments, CauMon always outperforms PCauM, with an improvement of at least 17.5% (in φ_4^{AT}). In particular, in those complex specifications that have nested temporal operators, such as φ_2^{AT}, φ_3^{AT} and φ_1^{NN}, PCauM can take extremely long time to monitor the traces: for φ_2^{AT} and φ_3^{AT}, it takes around 1900 s; for φ_1^{NN} that has nested operators of three levels, it even gets timeout. In the case of φ_2^{AT} and φ_3^{AT}, each instant of the signals takes about $\frac{1900}{30\times 100} = 0.63$ seconds, which is 63 times longer than the sampling period (0.01 s for AT) of the traces. This performance can lead to severe delays if PCauM is deployed in practice, and the main reason is, as shown in Example 2, because of the redundant computations of intermediate results. In contrast, CauMon does not suffer from the problem and provides a monitoring performance that allows its usage in a real-world setting, possibly even a synchronous monitoring setting.

The performance of PCauM is severely subject to the complexity of the specification. In the problems that have simpler formulas (e.g., φ_1^{AT}, φ_1^{AFC}), PCauM is not very slow. However, for specifications that have nested operators (e.g., φ_2^{AT}, φ_3^{AT}, φ_1^{NN}), PCauM becomes not feasible in practice. By contrast, CauMon suffers much less from this issue. Even for complex specifications, CauMon is still very efficient and its performance is always comparable with RobM.

By the comparison between RobM and CauMon, we observe that, the performance of CauMon is at the same magnitude of RobM, and so its performance

Table 3. Experimental results of the three monitors RobM, PCauM and CauMon Time is reported in seconds.

φ_1^{AT}	RobM	PCauM	CauMon	CauMon stat. (%)	
				Δ RobM	Δ PCauM
#1	0.34	0.31	0.20	−41.8	−35.5
#2	0.32	0.31	0.19	−40.2	−38.7
#3	0.30	0.3	0.18	−40.4	−40.0
#4	0.33	0.31	0.19	−40.6	−38.7
#5	0.39	0.34	0.23	−41.6	−32.4
#6	0.30	0.30	0.18	−40.1	−40.0
#7	0.32	0.31	0.19	−40.7	−38.7
#8	0.34	0.32	0.20	−41.0	−37.5
#9	0.31	0.31	0.19	−39.9	−38.7
#10	0.33	0.32	0.20	−40.9	−37.5

φ_2^{AT}	RobM	PCauM	CauMon	CauMon stat. (%)	
				Δ RobM	Δ PCauM
#1	1.55	1972.82	1.99	+28.4	−99.9
#2	1.50	1907.76	1.87	+24.9	−99.9
#3	1.47	1932.85	1.87	+27.5	−99.9
#4	1.48	1914.46	1.84	+24.8	−99.9
#5	1.50	1902.60	1.88	+25.8	−99.9
#6	1.48	1887.34	1.86	+25.7	−99.9
#7	1.45	1891.06	1.82	+25.7	−99.9
#8	1.44	1847.76	1.82	+26.2	−99.9
#9	1.58	1865.24	1.91	+21.0	−99.9
#10	1.41	1855.63	1.81	+28.2	−99.9

φ_3^{AT}	RobM	PCauM	CauMon	CauMon stat. (%)	
				Δ RobM	Δ PCauM
#1	1.63	2007.99	2.26	+39.1	−99.9
#2	1.64	2015.69	2.27	+38.2	−99.9
#3	1.63	1990.24	2.13	+30.5	−99.9
#4	1.59	1926.94	2.20	+38.2	−99.9
#5	1.62	1985.16	2.20	+35.8	−99.9
#6	1.64	2023.44	2.27	+38.4	−99.9
#7	1.57	1979.15	1.99	+27.0	−99.9
#8	1.57	1894.74	2.16	+37.9	−99.9
#9	1.63	1991.59	2.11	+29.2	−99.9
#10	1.61	1997.21	2.16	+33.9	−99.9

φ_4^{AT}	RobM	PCauM	CauMon	CauMon stat. (%)	
				Δ RobM	Δ PCauM
#1	0.46	0.57	0.47	+1.5	−17.5
#2	0.41	0.53	0.42	+2.4	−20.8
#3	0.45	0.57	0.46	+2.2	−19.3
#4	0.41	0.53	0.42	+2.3	−20.8
#5	0.43	0.55	0.44	+2.1	−20.0
#6	0.44	0.56	0.45	+1.4	−19.6
#7	0.42	0.54	0.43	+1.1	−20.4
#8	0.41	0.54	0.42	+1.4	−22.2
#9	0.43	0.55	0.43	+1.0	−21.8
#10	0.46	0.57	0.46	+0.5	−19.3

φ_1^{AFC}	RobM	PCauM	CauMon	CauMon stat. (%)	
				Δ RobM	Δ PCauM
#1	0.0079	0.0095	0.0056	−29.8	−41.1
#2	0.0066	0.0088	0.0048	−27.1	−45.5
#3	0.0063	0.0086	0.0047	−25.1	−45.3
#4	0.0061	0.0086	0.0047	−23.3	−45.3
#5	0.0061	0.0086	0.0047	−23.1	−45.3
#6	0.0061	0.0086	0.0046	−23.8	−46.5
#7	0.0061	0.0086	0.0047	−23.7	−45.3
#8	0.0061	0.0086	0.0047	−23.6	−45.3
#9	0.0061	0.0086	0.0047	−23.6	−45.3
#10	0.0062	0.0087	0.0047	−24.1	−46.0

φ_2^{AFC}	RobM	PCauM	CauMon	CauMon stat. (%)	
				Δ RobM	Δ PCauM
#1	0.0084	0.7033	0.0087	+3.1	−98.8
#2	0.0072	0.7041	0.0078	+8.0	−98.9
#3	0.0069	0.7017	0.0077	+12.0	−98.9
#4	0.0066	0.7041	0.0076	+15.0	−98.9
#5	0.0067	0.702	0.0077	+15.1	−98.9
#6	0.0066	0.7058	0.0076	+15.0	−98.9
#7	0.0067	0.7041	0.0077	+14.8	−98.9
#8	0.0067	0.7053	0.0077	+16.1	−98.9
#9	0.0067	0.7052	0.0077	+15.4	−98.9
#10	0.0067	0.7051	0.0078	+15.8	−98.9

φ_1^{NN}	RobM	PCauM	CauMon	CauMon stat. (%)	
				Δ RobM	Δ PCauM
#1	1.25	t/o	1.89	+51.4	−99.9
#2	1.22	t/o	1.86	+52.8	−99.9
#3	1.23	t/o	1.89	+53.4	−99.9
#4	1.22	t/o	1.88	+53.7	−99.9
#5	1.23	t/o	1.88	+52.7	−99.9
#6	1.22	t/o	1.88	+53.8	−99.9
#7	1.23	t/o	1.88	+53.0	−99.9
#8	1.24	t/o	1.89	+52.8	−99.9
#9	1.24	t/o	1.88	+52.3	−99.9
#10	1.23	t/o	1.86	+51.4	−99.9

φ_1^{FFR}	RobM	PCauM	CauMon	CauMon stat. (%)	
				Δ RobM	Δ PCauM
#1	0.053	850.1	0.085	+60.7	−99.9
#2	0.049	857.6	0.080	+62.3	−99.9
#3	0.049	857.1	0.080	+63.0	−99.9
#4	0.051	822.0	0.082	+61.7	−99.9
#5	0.049	813.5	0.080	+62.8	−99.9
#6	0.050	822.0	0.081	+63.3	−99.9
#7	0.051	867.8	0.083	+61.4	−99.9
#8	0.047	809.0	0.077	+62.3	−99.9
#9	0.047	809.6	0.077	+64.0	−99.9
#10	0.047	809.4	0.077	+63.1	−99.9

is comparable with RobM. While in some cases CauMon is not as fast as RobM, the performance difference is not very large. This is acceptable, regarding that our CauMon can provide more information about system evolution than RobM. There also happens that CauMon is faster than RobM, with simple specifications that have no nested temporal operators, such as φ_1^{AT} and φ_1^{AFC}. This is because monitoring simple specifications, like $\Box_I \alpha$, mainly needs the causation distance lists $\mathsf{Cau}^{\ominus}[\alpha]$ and $\mathsf{Cau}^{\oplus}[\alpha]$ of atomic proposition α, and by Defs. 3 and 4, $\mathsf{Cau}^{\ominus}[\alpha]$ and $\mathsf{Cau}^{\oplus}[\alpha]$ have simpler shape than $\mathsf{Rob}^{\mathsf{U}}[\alpha]$ and $\mathsf{Rob}^{\mathsf{L}}[\alpha]$.

7 Conclusion and Future Work

We propose an efficient approach for online causation monitoring. Our approach features two dynamic programming strategies, namely, the use of causation distance lists that record intermediate results, and the use of sliding window algorithms that accelerate the causation computation of temporal operators. Experiments show that, in terms of efficiency, our approach significantly outperforms the plain causation monitor in [26], and is comparable with the existing online robust monitors that deliver less information about system evolution than ours.

As future work, we would like to explore the application of the proposed monitors for system behavior analysis. For instance, by causation monitoring, we can obtain the information about when a cause of the specification violation happens, and this can be used for fault analysis such as localization and repair.

Acknowledgments. Z. Zhang is supported by JSPS KAKENHI Grant No. JP23K16865 and No. JP23H03372. P. Arcaini is supported by Engineerable AI Techniques for Practical Applications of High-Quality Machine Learning-based Systems Project (Grant Number JPMJMI20B8), JST-Mirai. J. An and I. Hasuo are supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST, Funding Reference number: 10.13039/501100009024 ERATO.

Data Availability Statement. All relevant data that support the findings of this paper are available in Zenodo with the identifier https://doi.org/10.5281/zenodo.12518433.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TaLiRo: a tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 254–257. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_21
2. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification. LNCS, vol. 10457. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5
3. Bartocci, E., Ferrère, T., Manjunath, N., Ničković, D.: Localizing faults in Simulink/Stateflow models with STL. In: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week), pp. 197–206. HSCC 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3178126.3178131
4. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 260–272. Springer, Heidelberg (2006). https://doi.org/10.1007/11944836_25
5. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 1–64 (2011). https://doi.org/10.1145/2000799.2000800
6. Deshmukh, J.V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., Seshia, S.A.: Robust online monitoring of signal temporal logic. Formal Methods Syst. Des. **51**(1), 5–30 (2017). https://doi.org/10.1007/s10703-017-0286-7
7. Dokhanchi, A., Hoxha, B., Fainekos, G.: On-line monitoring for temporal logic robustness. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 231–246. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_19
8. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_17
9. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15297-9_9
10. Ernst, G., et al.: ARCH-COMP 2021 category report: falsification with validation of results. In: Frehse, G., Althoff, M. (eds.) 8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21). EPiC Series in Computing, vol. 80, pp. 133–152. EasyChair (2021). https://doi.org/10.29007/xwl1
11. Ernst, G., et al.: ARCH-COMP 2020 category report: falsification. In: Frehse, G., Althoff, M. (eds.) 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20). EPiC Series in Computing, vol. 74, pp. 140–152. EasyChair (2020). https://doi.org/10.29007/trr1
12. Ernst, G., et al.: ARCH-COMP 2022 category report: falsification with Ubounded resources. In: Frehse, G., Althoff, M., Schoitsch, E., Guiochet, J. (eds.) Proceedings of 9th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH22). EPiC Series in Computing, vol. 90, pp. 204–221. EasyChair (2022). https://doi.org/10.29007/fhnk
13. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoret. Comput. Sci. **410**(42), 4262–4291 (2009). https://doi.org/10.1016/j.tcs.2009.06.021

14. Ho, H.-M., Ouaknine, J., Worrell, J.: Online monitoring of metric temporal logic. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 178–192. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_15
15. Jakšić, S., Bartocci, E., Grosu, R., Kloibhofer, R., Nguyen, T., Ničkovié, D.: From signal temporal logic to FPGA monitors. In: Proceedings of the 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign, pp. 218–227. MEMOCODE 2015, IEEE Computer Society, USA (2015). https://doi.org/10.1109/MEMCOD.2015.7340489
16. Jakšić, S., Bartocci, E., Grosu, R., Nguyen, T., Ničković, D.: Quantitative monitoring of STL with edit distance. Formal Methods Syst. Des. **53**, 83–112 (2018). https://doi.org/10.1007/s10703-018-0319-x
17. Jin, X., Deshmukh, J.V., Kapinski, J., Ueda, K., Butts, K.: Powertrain control verification benchmark. In: Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control, pp. 253–262. HSCC 2014, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2562059.2562140
18. Lemire, D.: Streaming maximum-minimum filter using no more than three comparisons per element. Nordic J. Comput. **13**(4), 328–339 (2006)
19. Lima, L., Herasimau, A., Raszyk, M., Traytel, D., Yuan, S.: Explainable online monitoring of metric temporal logic. In: Sankaranarayanan, S., Sharygina, N. (eds.) International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 13994, pp. 473–491. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30820-8_28
20. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
21. Menghi, C., et al.: ARCH-COMP23 category report: Falsification. In: Frehse, G., Althoff, M. (eds.) Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH23). EPiC Series in Computing, vol. 96, pp. 151–169. EasyChair (2023). https://doi.org/10.29007/6nqs
22. Nickovic, D., Maler, O.: AMT: A property-based monitoring tool for analog systems. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 304–319. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75454-1_22
23. Ničković, D., Yamaguchi, T.: RTAMT: online robustness monitors from STL. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 564–571. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_34
24. Selyunin, K., et al.: Runtime monitoring with recovery of the SENT communication protocol. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 336–355. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_17
25. Ulus, D.: Online monitoring of metric temporal logic using sequential networks. arXiv preprint arXiv:1901.00175 (2019)
26. Zhang, Z., An, J., Arcaini, P., Hasuo, I.: Online causation monitoring of signal temporal logic. In: Enea, C., Lal, A. (eds.) Computer Aided Verification, pp. 62–84. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-37706-8_4
27. Zhang, Z., Arcaini, P., Xie, X.: Online reset for signal temporal logic monitoring. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **41**(11), 4421–4432 (2022). https://doi.org/10.1109/TCAD.2022.3197693

28. Zhang, Z., Ernst, G., Sedwards, S., Arcaini, P., Hasuo, I.: Two-layered falsification of hybrid systems guided by monte Carlo tree search. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(11), 2894–2905 (2018)
29. Zhang, Z., Hasuo, I., Arcaini, P.: Multi-armed bandits for Boolean connectives in hybrid system falsification. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 401–420. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_23
30. Zhang, Z., Lyu, D., Arcaini, P., Ma, L., Hasuo, I., Zhao, J.: Effective hybrid system falsification using monte Carlo tree search guided by QB-robustness. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 595–618. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_29

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Industry Day Track

UnsafeCop: Towards Memory Safety for Real-World *Unsafe* Rust *Code* with *P*ractical Bounded Model Checking

Minghua Wang[1](✉), Jingling Xue[2,3], Lin Huang[1], Yuan Zi[1,4], and Tao Wei[1]

[1] Ant Group, Beijing, China
{minghua.wmh,linyu.hl,ziyuan.zy,lenx.wei}@antgroup.com, ziyuan@stu.pku.edu.cn
[2] Ant Group, Sydney, Australia
[3] UNSW Sydney, Sydney, Australia
j.xue@unsw.edu.au
[4] Peking University, Beijing, China

Abstract. Rust has gained popularity as a safer alternative to C/C++ for low-level programming due to its memory-safety features and minimal runtime overhead. However, the use of the "unsafe" keyword allows developers to bypass safety guarantees, posing memory-safety risks. Bounded Model Checking (BMC) is commonly used to detect memory-safety problems, but it has limitations for large-scale programs, as it can only detect bugs within a bounded number of executions.

In this paper, we introduce *UnsafeCop* that utilizes and enhances BMC for analyzing memory safety in real-world unsafe Rust code. Our methodology incorporates harness design, loop bound inference, and both loop and function stubbing for comprehensive analysis. We optimize verification efficiency through a strategic function verification order, leveraging both types of stubbing. We conducted a case study on TECC (Trusted-Environment-based Cryptographic Computing), a proprietary framework consisting of 30,174 lines of Rust code, including 3,019 lines of unsafe Rust code, developed by Ant Group. Experimental results demonstrate that *UnsafeCop* effectively detects and verifies dozens of memory safety issues, reducing verification time by 73.71% compared to the traditional non-stubbing approach, highlighting its practical effectiveness.

Keywords: Unsafe Rust · Memory Safety · Bounded Model Checking

1 Introduction

Rust's emphasis on memory safety is well-recognized, with its rigorous ownership based type system effectively eliminating numerous memory-safety issues during compilation. As a result, critical systems [27,32] are increasingly being built from the ground up in Rust. However, Rust, while making substantial strides in enhancing memory safety, still permits developers to write unsafe code, which undermines its safety guarantees and provides developers with powerful but risky capabilities [31]. Recent studies [13,24,33] have highlighted that unsafe code remains the predominant source of memory-safety problems in Rust.

BMC (Bounded Model Checking) [15,16,25] is a widely used technique for verifying memory-safety properties in unsafe Rust. It encodes program traces as symbolic SAT/SMT problems and employs solvers to provide bounded proofs. However, BMC has limitations, including the need for a fixed number of program executions, which requires setting bounds on loop iterations. Smaller loop bounds may result in incomplete unwinding, potentially missing genuine bugs. On the other hand, overly large bounds can lead to memory exhaustion and termination of the checker. Additionally, BMC's effectiveness is constrained when dealing with complex code, especially paths involving intricate functions, as the generated formulas can become too complex for practical solver handling. These challenges collectively impede BMC's utility in real-world code verification.

In this paper, we present *UnsafeCop*, which utilizes and enhances Kani [15], a bounded model checker, to verify memory safety in real-world unsafe Rust code. Our approach identifies functions that execute unsafe code and generates proof harnesses with test cases. Abstract interpretation determines loop bounds for BMC, and we implement stubbing for loops with large bounds to maintain essential safety checks and ensure soundness. Our BMC model checking uses a scheduling strategy that prioritizes complex, frequently invoked functions, using stubs for others to increase verification efficiency. We validated *UnsafeCop*'s effectiveness through a case study on the TECC (Trusted-Environment-based Cryptographic Computing) framework, a proprietary software by Ant Group that includes 30,174 lines of Rust code, with 3,019 lines of unsafe Rust code. TECC integrates trusted computing with secure multi-party computation techniques to foster a secure, reliable, and high-performance computing environment for large-scale data applications. This case study served as a robust test environment for assessing *UnsafeCop*'s memory safety capabilities.

In summary, this paper makes the following three main contributions:

– A practical BMC approach for detecting memory-safety issues in Rust programs, which includes harness design, loop bound inference, stubbing complex loops, and optimizing function verification order by utilizing function stubbing for improved performance.
– The evaluation of *UnsafeCop* on a real-world project comprising 30,174 lines of Rust code, with 3,019 lines being written in unsafe Rust.
– Insights and lessons learned from verifying real-world Rust programs, along with suggestions for improving current model-checking tools such as Kani [15], particularly for Rust.

2 Related Work

Program analysis [7,17,18,22,23] can identify memory-safety bugs in Rust code but do not guarantee soundness. Formal methods, including theorem prov-

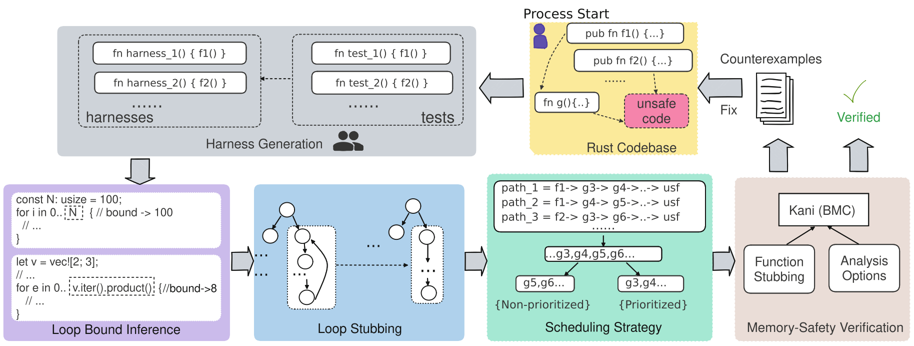


Fig. 1. Architecture of *UnsafeCop*.

ing [14,20,21,30] and deductive proving [2,8], are used to verify functional properties in safe Rust. Techniques like abstract interpretation [10,22], symbolic execution [19,26,28], and bounded model checking (BMC) [15,16,25] are focused on ensuring memory safety properties. BMC stands out by encoding program traces into symbolic SAT/SMT problems for solver-based automatic verification. Addressing challenges like appropriate loop bounds and managing path explosion is essential for BMC to be practically applicable in large-scale codebase verification.

3 Overview

We introduce *UnsafeCop*, a method for verifying memory safety in real-world unsafe Rust code. Our approach, depicted in Fig. 1, starts by locating all unsafe code in the project and identifying functions exposed to other crates that execute unsafe code. Test cases for these exposed functions are transformed into proof harnesses. We infer loop bounds for accurate program modeling and apply loop stubbing with memory safety checks for loops with large bounds. Using Kani [15], a bit-precise BMC for Rust, we perform model checking on TECC. Our strategy optimizes function verification order, prioritizing high-complexity, frequently-invoked functions, followed by stubbing verified functions to check the rest. Kani generates counterexamples for any detected memory issues, which are iteratively fixed. Our focus is on memory safety in user-provided unsafe code, excluding unsafe code from the Rust standard library and third-party crates.

3.1 Verification Scope and Proof Harness

In the verification model detailed in Def-1 below, the scope of verification, designated as TF, includes the set of functions necessitating proof harnesses. This set encompasses public functions that have the capability to access user-provided unsafe code, identified as USF, which includes unsafe blocks, unsafe functions, or interfaces using Foreign Function Interfaces (FFI). A function f is classified as public to other crates if it $\mathit{isPublic}(f) = \mathit{true}$. Furthermore, the function $\mathit{Reach}(f, \mathit{usf})$ assesses whether there is a program path that allows function f to access any unsafe code usf, determined via reachability analysis on the program's interprocedural Control Flow Graph (iCFG) as is standard.

$$TF = \{f \mid \mathit{isPublic}(f) \ \&\& \ \mathit{Reach}(f, \mathit{usf}), \mathit{usf} \in \mathit{USF}\} \tag{Def-1}$$

To ensure memory safety in unsafe Rust, it is essential to create and verify harnesses for functions within the set TF. The process of developing a harness for a target public function is carried out in three stages.

First, the harness must encompass all possible calling scenarios of the target public function. This is effectively achieved by leveraging existing test cases, including both integration and unit tests available within the project. These tests, carefully crafted by developers, not only demonstrate how the target public function is used but also ensure necessary initializations and data setups are performed before the function is invoked. If the target public function lacks sufficient tests, we collaborate closely with developers to design a harness that accurately reflects real-world invoking scenarios.

Additionally, if the target public function uses generic types, either in its parameters or within the function itself, which can significantly alter control flow, we collaborate with developers to identify all appropriate concrete types. It is crucial for the harness to explore all possible instantiations of these generics to capture varied control flow paths, ensuring thorough testing and verification.

Finally, it is essential for the harnesses to ensure thorough code coverage. This coverage might change due to code adjustments when memory safety issues are identified during verification. After resolving all detected issues, if code coverage is still found to be insufficient, collaborating with developers to adjust the range of values for symbolic variables may be necessary to achieve more comprehensive coverage. A harness is deemed correctly generated when the associated public function is verified to be free of bugs and achieves sufficient coverage.

3.2 Loop Bound Inference

BMC offers bounded proofs, ensuring that within given loop bounds, the program satisfies certain properties. However, if the loop bound is too small, it may lead to semantic deviations from the actual program, risking missed memory safety bugs. Setting an appropriate loop bound is key for maintaining verification soundness and detecting memory safety issues.

We utilize abstract interpretation [6] to deduce loop bounds. Observations indicate that many loops, like those in the first two code snippets of Fig. 2, have identifiable patterns (i.e., clear signatures). In the first example, the loop's upper bound is a literal constant directly. In the second example, the loop counters rely on variable values derived from constants or intervals. Here, we calculate each loop counter's interval using the interval domain [5] in abstract interpretation, with the interval's upper bound serving as the loop bound.

For loops lacking clear patterns (i.e., clear signatures), as in the third code snippet, we infer loop bounds using widening in the interval domain, achieving a

```
 1  /* code snippet 1 */
 2  for _ in 0..T::WIDTH { // i.e., 32
 3      x0_.push(T::new(dim, shape));
 4  }
 5  /* code snippet 2 */
 6  let nr_rnd = T::WIDTH.log2();
 7  for rnd in 0..nr_rnd {  // loop 1
 8      let rm_bit = 1 << (nr_rnd - rnd);
 9      for i in 0..rm_bit { // loop 2
10          if i & 1 == 1 {...}
11      }
12  }
```

```
 1  /* code snippet 3 */
 2  let elem = kani::any_where(|x|
 3      *x <=12 && *x >=1);
 4  let a = vec![elem; 8];
 5  let mut i = 0;
 6  loop { // no clear loop signature
 7      if i >= a.len() { break; }
 8      if i * 2 > a[i] { break; }
 9      i += 1;
10  }
```

Fig. 2. Loop bound inference. Snippet 1 (Color figure online) demonstrates a loop with the constant bound `T::WIDTH`. In snippet 2, values of `nr_rnd` and `rm_bit` are derived from other constants. In snippet 3, the loop counter is determined through widening within the interval domain.

Table 1. Intervals of the loop index `i` for the third code snippet in Fig. 2 at each iteration, with the fixed-point algorithm set to iterate for four times.

Line	Iteration 1	Iteration 2	Iteration 3	Iteration 4
7	[1, 1]	[2, 2]	[3, 3]	[3, 7]
8	[1, 1]	[2, 2]	[3, 3]	[3, 7]

fixed-point in the analysis. This involves setting a predefined number of iterations for the abstract interpretation's fixed-point algorithm. As the loop progresses, we monitor the intervals of accessed variables. Upon completing the final iteration, these variables are widened to a fixed-point state. The upper bound of the widened interval for a loop index variable becomes the loop bound. Table 1 shows the loop index `i`'s intervals at each iteration, with the algorithm iterating four times, resulting in a final loop bound of 7.

3.3 Loop Stubbing

Some loops yield excessively large bounds from widening, making complete unwinding infeasible for BMC, which might lead to out-of-memory issues when generating verification conditions. To address this, we apply loop stubbing to rewrite loops with unfeasible bounds for unwinding.

We present loop stubbing guidelines using the example depicted in Fig. 3:

Iterator-like Variables. For iterator-like variables, such as loop index variables and references to elements in array-like structures, we substitute their values with symbolic intervals. As demonstrated in Fig. 3, the iterator i accesses vectors like `a.0` and `a.1`, with their sizes represented by the interval `[0, a.size())`.

```
pub fn add <T:AType> (a: &TView<T>, b: &TView<T>, c: &mut TViewMut<T>) {
 if a.size() != b.size() || a.size() != c.size() { return; }
 //a.0, a.1, b.0, b.1, c.0 and c.1 are vectors. These vector's loop bounds are a.size()
 for (x, xu, y, yu, r, ru) in izip!(a.0.iter(), a.1.iter(),
       b.0.iter(),  b.1.iter(), c.0.iter_mut(), c.1.iter_mut()) {
    *r = *x + *y;
    *ru = *xu + *yu;
  }
}
```

```
pub fn add_loop_stubbed <T:AType> (a: &TView<T>, b: &TView<T>, c: &mut TViewMut<T>) {
 if a.size() != b.size() || a.size() != c.size() { return; }
 // ensure that the loop counters are all correct
 assert!(a.size() == b.size() && b.size() == c.size());
 // intervalize the iterator-like variables with Kani primitive functions
 let i: usize = kani::any_where(|i_| *i_ < a.size());
 let (rx, rxu, ry, ryu, rr, rru) = (&a.0[i], &a.1[i], &b.0[i], &b.1[i],
        &mut c.0[i], &mut c.1[i]);
 let (x, xu, y, yu, r, ru)=(*rx, *rxu, *ry, *ryu, *rr, *rru); //deref checks maintained
 r = x + y; ru = xu + yu; //overflow checks maintained
 // over-approximate side-effects
 *rr = kani::any_where(|v: &T| *v >= min_T && *v <= max_T);
 *rru = kani::any_where(|v: &T| *v >= min_T && *v <= max_T);
}
```

Fig. 3. Loop stubbing (top: original code; bottom: loop-stubbed code).

Additionally, new references like `rx` and `rxu` are introduced to symbolize the iterators of accessed vectors, such as `&a.0[i]`, `&a.1[i]`.

Memory-Safety Properties. For array-like structures, we use `assert` for access validation, as demonstrated in Fig. 3. For example, the assertion `assert!(a.size()==b.size() && b.size()==c.size())` ensures accesses to vectors remain within their boundaries. Based on the new references, `rx`, `rxu`, `ry`, `ryu`, `rr`, and `rru` introduced above, statements `*r = *x + *y` and `*ru = *xu + *yu` are replaced with `r = x + y` and `ru = xu + yu`, respectively. Dereference and arithmetic overflow checks are maintained.

Side-Effect Over-Approximation. Side effects arise from write operations inside a loop affecting variables declared outside it. To address this, we over-approximate these variables by assigning them appropriate intervals. For example, the elements in `c.0` and `c.1`, which are overwritten within the loop in Fig. 3, can assume any value of type `T`. With `rr` and `rru` being iterator-like symbolic variables, the last two assignments result in the specified over-approximation.

When a loop contains function calls, we assess the callee functions for complexity and call frequency. Functions that are complex and frequently called are verified first according to our scheduling strategy and replaced with stubs in subsequent analyses. Simpler functions, on the other hand, undergo standard analysis. We will provide more details on this approach in Sect. 3.4.

In loop stubbing, we preserve memory-safety checks and over-approximate loops' external side effects to prevent BMC from stalling, enabling thorough analysis. Over-approximation is automated, whereas safety check preservation is manual, ensuring no memory safety issues are missed and maintaining soundness.

3.4 Scheduling Strategy

The time spent on constraint solving is significant. The order in which functions are verified affects both the quantity and complexity of the generated verification conditions. This, in turn, influences the duration of constraint solving and the overall performance of the verification process.

We use Def-2 to denote a specific path analyzed when verifying a harness:

$$\mathit{VerifyPath}_i = t \rightarrow f_1 \rightarrow ... \rightarrow f_n \rightarrow \mathit{usf}_i, \quad \text{where } t \in TF \text{ and } \mathit{usf}_i \in \mathit{USF} \tag{Def-2}$$

where t represents a target public function undergoing verification, usf_i refers to reachable unsafe code from t, and $f, ..., f_n$ are n additional functions executed along the analyzed path. For recursion, a bounded depth of 1 is used. To ensure the absence of memory safety issues, all paths starting from a harness must be explored. The straightforward but inefficient approach of verifying all the paths in a function individually is recognized. When a specific function f, particularly one with complex logic, recurs across multiple paths, it can significantly affect the time allocated for constraint solving in the program's overall verification process, due to the unnecessary multiple analyses of f.

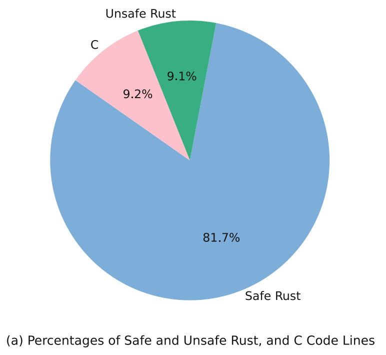
We employ a scheduling strategy to optimize the order of function verification. For each path outlined in Def-2, if a function f is frequently invoked and sufficiently complex, we include it in a set, F_{sche}, as defined in Def-3 below:

$$F_{sche} = \{f \mid \mathit{Invk}(f) \times \mathit{Cmplx}(f) > T\_\mathit{invk}\_\mathit{cmplx}\} \tag{Def-3}$$

The "verification complexity" of f is determined by multiplying its invocation frequency, denoted as $\mathit{Invk}(f)$, with its computational complexity, referred to as $\mathit{Cmplx}(f)$. We use a predefined threshold, $T\_\mathit{invk}\_\mathit{cmplx}$, for this calculation.

$\mathit{Invk}(f)$ represents the in-degree of f in the program's iCFG. To determine $\mathit{Cmplx}(f)$, we take into account both the Halstead effort [11] and cyclomatic complexity [9]. Halstead effort focuses on understanding difficulty, accounting for code length, operations, and operators, while cyclomatic complexity addresses control flow complexity. Both metrics collectively indicate the complexity of a function. We utilize rust-code-analysis [1] to calculate both metrics and their product serves as an indicator of a function's complexity.

To execute this verification order, we first verify the functions in F_{sche}. Then, to verify the remaining functions, we substitute the original functions at their callsites with their respective stubs. Each stub is an over-approximation of its function, computed in a manner akin to a loop stub, as detailed in Sect. 3.3.



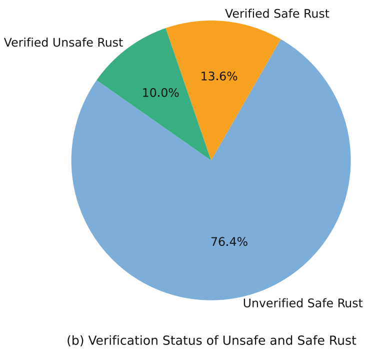


Fig. 4. Percentage breakdown of implemented and verified code w.r.t. memory safety.

3.5 Memory-Safety Verification

We use Kani-0.22.0 [15], compiled with the CadiCal solver [4], for model checking, employing the "--memory-safety-checks" option. We have not set a timeout for the solver. Loop unwinding information is provided using the "--cbmc-args --unwindset $L_1 : B_1$,$L_2 : B_2$ ..." option, where B_i indicates the inferred loop bound for loop L_i. In alignment with our scheduling strategy, the functions in F_{sche} were given verification priority. For the verification of subsequent functions, `kani::stub` was utilized to stub the already verified functions in F_{sche}.

4 Evaluation

TECC, short for Trusted-Environment-based Cryptographic Computing, is a proprietary framework developed by Ant Group. It integrates trusted computing with multi-party computation techniques, aiming to provide a secure, reliable, and high-performance computing environment for large data applications.

TECC consists of 3,060 lines of C and 30,174 lines of Rust, including 3,019 lines of unsafe Rust as shown in Fig. 4(a). Rust handles critical tensor computations, while C oversees computation algorithms that interface with Rust via FFI. The unsafe Rust includes 68 unsafe blocks across 27 functions (351 lines), 6 unsafe functions (106 lines), and 96 FFI functions (2,562 lines).

To demonstrate *UnsafeCop*'s capability in identifying memory-safety issues, we applied it to verify TECC in a major case study. We dedicated approximately 115 person-hours to verifying 7,118 lines of Rust code, which includes 3,019 lines of unsafe code and 4,099 lines of safe code. The percentages of these lines relative to the total Rust code (excluding the C code) are shown in Fig. 4(b).

We employed Kani-0.22.0 [15] for Bounded Model Checking (BMC) on our TECC project. Kani serves as a backend for the Rust compiler and utilizes the C Bounded Model Checker (CBMC) [16] as its verification engine. It is specifically designed to target Rust's Mid-level Intermediate Representation (MIR), which we used for our interval analysis on MIR. Before initiating verification of the

```
// Generic T: A64
// Generic B: B64
impl Executor {
 pub fn lt_zero_less_turn
 <T: AType, B: BType>(
  &mut self,
  x: &Tensor<T>,
  r: &mut Tensor<B>)
   -> Result<(), TError>
  {
    //...
    // will call unsafe
  }
}
```

```
pub unsafe extern "C" fn
ffi_lt_zero_less_turn_A64_B64(exec_p: *mut c_void,
 x: *mut CFFITensor, r: *mut CFFITensor) -> i32 {
 match catch_unwind(move || {
  if exec_p.is_null() { panic!("exec_p␣is␣null"); }
  let exec = &mut *(exec_p as *mut Executor);
  let x = tensor_from_c::<A64>(x).unwrap();
  let mut r = tensor_from_c::<B64>(r).unwrap();
  exec.lt_zero_less_turn(&x, &mut r).unwrap()
 }) {
    Err(e) => -1,
    Ok(r) => 0,
 }
}
```

Fig. 5. Public function `lt_zero_less_turn<A64, B64>` and its FFI wrapper `ffi_lt_zero_less_turn_A64_B64`, with both sharing the same inputs.

Rust code, we resolved undefined behaviors and memory safety issues in the C code using the TrustInSoft Analyzer [29]. Additionally, all C functions callable from Rust were stubbed to over-approximate their side effects.

The evaluation was conducted using an Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz with 16 cores and 128GB RAM, running Ubuntu 18.04.6 LTS.

4.1 Harness Design

Below, we outline the process involved in harness design in verifying TECC.
Verification Scope. In TECC, 243 public functions have the capability to access unsafe code. Of these, developers confirmed that 110 are accessible to other crates. Among these 110, 96 have corresponding FFI wrappers, as shown in Fig. 5. Both the FFI functions and the public functions they wrap receive the same inputs; however, the FFI functions are more prone to memory safety issues as they directly receive data from C code. Consequently, we focused on developing specific harnesses for these FFI functions, as depicted in Fig. 6.

Harness Writing. TECC features 57 integration and unit tests, covering 89 of the 110 public functions that require verification harnesses. We converted these existing tests into harnesses and developed 21 new ones to achieve full coverage of all necessary functions. The creation of these 78 harnesses consumed 20% of the total verification effort, translating to approximately 23 person-hours.

During the harness development process, we collaborated with TECC developers to remove 14 public functions with access to unsafe code from the codebase. These functions were deemed unlikely to be called externally, thus reducing the potential attack surface associated with unsafe code.

Coverage Statistics. For each public function harness in TECC, we considered all potential calling scenarios, covering all possible values for symbolic variables to ensure thorough code coverage. On average, two rounds of discussions with

```
// a helper function to generate a non-deterministic vector
fn gen_nondet_vec<T>(sz: usize) -> Vec<T> where T: kani::Arbitrary {
 let mut v = Vec::<T>::with_capacity(sz);
 unsafe {v.set_len(sz); std::ptr::write_bytes(v.as_mut_ptr(), u8::any(), sz);}
 v
}
// a helper function to generate a non-deterministic CFFITensor
fn gen_nondet_cffitensor<T>(dim: usize, elem: usize) -> CFFITensor
where T: kani::Arbitrary {
 let shape = vec![elem; dim];
 let prod = shape.iter().product::<usize>();
 let (mut d, mut du) = (gen_nondet_vec::<T>(prod), gen_nondet_vec::<T>(prod));
 CFFITensor { size: prod, data: d.as_mut_ptr() as *mut c_void,
        data_u: du.as_mut_ptr() as *mut c_void }
}
// a harness for ffi_lt_zero_less_turn_A64_B64
fn harness() {
 let exec *mut c_void = get_any_exec();
 let dim: usize = kani::any_where(|x| *x <= 3);
 let elem: usize = kani::any_where(|x| *x <= 1,000);
 let mut x = gen_nondet_cffitensor::<A64>(dim, elem);
 let mut r = gen_nondet_cffitensor::<B64>(1, (x.size + B64::N_BITS-1)/B64::N_BITS);
 unsafe {ffi_lt_zero_less_turn_A64_B64(exec, &mut x, &mut r)}
}
```

Fig. 6. Harness development for the FFI function given in Fig. 5 `ffi_lt_zero_less_turn_A64_B64` instead of its non-FFI version `lt_zero_less_turn<A64,B64>`, as FFI functions are more prone to memory safety problems.

developers were conducted for each harness to confirm the calling scenarios and appropriate ranges for symbolic variables. Ultimately, we achieved approximately 95% statement coverage for all verified functions, resolving 39 identified memory-safety issues through 19 rounds of discussions with developers. Throughout the verification process, there were no adjustments to the value ranges of symbolic variables to expand code coverage.

Vacuity Checks. The correctness of each harness was ensured through consultations with developers and complemented by automatic vacuity checks to confirm property reachability using Kani. Properties identified as vacuous were marked as `UNREACHABLE` in Kani's output. We specifically focused on these properties, which represented less than 0.3% of all verified properties. Our review confirmed that there were no possible traces that could reach these properties given the determined data setups in our harnesses.

4.2 Improvements on Verification Efficiency

We demonstrate the improvements achieved by applying loop stubbing combined with our scheduling strategy that also incorporates function stubbing.

Loop Stubbing. We illustrate the performance improvement achieved through loop stubbing using the TECC function `add`, as previously shown in Fig. 3.

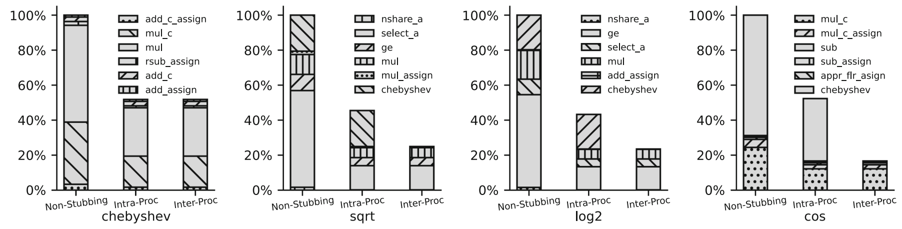


Fig. 7. Verification times of Non-Stubbing, Intra-Proc and Inter-Proc for four functions selected in TECC (with Non-Stubbing and Intra-Proc defined in Sect. 4.2).

According to developers, the practical loop bound `a.size()` can reach 100 million. Without loop stubbing, we set this loop bound and attempted verification, resulting in Kani spending hours in the pre-processing step before being terminated due to running out of memory. In contrast, after applying loop stubbing, Kani successfully produced the desired verification result in 6.75 h.

Scheduling Strategy. Fig. 7 illustrates the performance advantages of our scheduling strategy, introduced in Sect. 3.4, denoted as Inter-Proc. This strategy is compared to two simpler approaches, Non-Stubbing and Intra-Proc, for four functions within TECC. Non-Stubbing involves the individual analysis of all functions without the use of stubs to prevent re-analysis of callee functions. Intra-Proc conducts individual function analysis but utilizes stubs to avoid redundant analysis of the same callee function called from within the same function.

Compared to Non-Stubbing, Intra-Proc reduces the average verification time by 51.76% for the four functions: `chebyshev`, `sqrt`, `log2`, and `cos`. In the case of `chebyshev`, Inter-Proc and Intra-Proc yield the same performance. On average, Inter-Proc improves performance over Non-Stubbing by 70.75%. When considering only `sqrt`, `log2`, and `cos`, the average performance gain is 78.28%.

When expanding our evaluation to the entire TECC codebase, it took us around 115 h to verify 110 public functions with Inter-Proc, which accesses unsafe Rust code. During this process, we addressed and fixed the 39 reported bugs (detailed in Sect. 4.3). In contrast, we estimate that Non-Stubbing would require around 437 person-hours to accomplish the same verification task. To assess the performance improvement of Inter-Proc over Non-Stubbing, we estimate the verification times spent for both scheduling strategies as follows:

$$T_{Non-Stubbing} = \sum_{f \in F_{exam}} \mathit{Invk}(f) \times \mathit{Cmplx}(f) \tag{Def-4}$$

$$T_{Inter-Proc} = \sum_{f \in F_{sche}} 1 \times \mathit{Cmplx}(f) + \sum_{f \in F_{exam} - F_{sche}} \mathit{Invk}(f) \times \mathit{Cmplx}(f) \tag{Def-5}$$

where F_{exam} denotes the set of functions examined by BMC. For the functions in F_{sche}, their invocation times are assumed to be 1, as each is verified once.

Based on our observations, some frequently called functions have low verification complexity, while others with complex logic are invoked infrequently.

In such cases, it is reasonable to skip stubbing these functions, as it would not significantly impact the overall verification time.

We computed the product of $Invk(f)$ and $Cmplx(f)$ for all functions in F_{exam} and utilized their harmonic mean [12] as the threshold T_invk_cmplx in Def-3 to eliminate functions with extremely low verification complexity. For TECC, we initially had $|F_{exam}| = 196$. By setting $T\_invk\_cmplx = 3360.16$, we obtained $|F_{sche}| = 168$ after filtering out 28 functions. The harmonic mean for F_{sche} is 20014.82. Ultimately, Inter-Proc is estimated to reduce overall verification time by approximately 73.71% compared to Non-Stubbing.

4.3 Effectiveness

In the verification of TECC, *UnsafeCop* detected a total of 39 memory-safety issues, as detailed in Table 2. All of these issues were confirmed and addressed by the developers. Subsequently, *UnsafeCop* verified their absence, ensuring that the identified memory safety problems had been effectively resolved.

Table 2. Memory safety problems detected in TECC by *UnsafeCop*.

Bug Type	Access Out-of-Bound	Arith. Overflow	Ptr Deref	Double Free	Mem Leak
Count	14	20	2	1	2

When performing stubbing to summarize loops and functions, we approximate their side effects using the interval domain. It is worth noting that this approach did not introduce any false positives, as TECC comprises loops and functions that operate on unrestricted values within their respective domains.

UnsafeCop comprises four main components: Harness Design (HD), Loop Bound Inference (LBI), Loop Stubbing (LS), and Inter-Proc Scheduling (IS), where function stubbing is also performed. These elements collectively contribute to practical verification efforts. Table 3 illustrates their roles in identifying memory-safety issues in TECC. HD is instrumental in detecting all bugs, with HD-O representing bugs exclusively identified by HD. Both LS and IS prove highly effective, detecting the majority of bugs and demonstrating their capabilities for in-depth analysis.

Table 3. Contributions of *UnsafeCop*'s four main components to its overall effectiveness in uncovering memory-safety issues in TECC.

UnsafeCop's Techniques	HD+IS+LS	HD+LBI	HD+LS	HD-O	HD+LBI+LS
#Mem-Safety Issues Found	11	2	15	10	1

Let us examine two case studies to see how *UnsafeCop* identifies two bugs.

```
pub fn truncate(tr: &[usize],
    x: &[usize], r: &mut [usize]) {
 let mut tmp = [0; x.len()];
 // loop bound: 100,000,000
 for (tt, x_) in izip!(
   tmp.iter_mut(), x.iter()) {
   *tt += *x_ * *x_;
 }
 //...
 for (r1p_, sub_, tmp_, tru_) in
  izip!(r.iter_mut(),
   r.iter(),
   tmp.iter(),
   tr.iter().chain(
    repeat(&tr[tr.len()-1])) //Arith.
      Overflow!
  ) { //...
  }
}
```

```
pub unsafe extern "C" fn ffi_truncate(
    sz: size_t, tr_p: *mut size_t,
    x_p: *mut size_t, r_p: *mut size_t) {
 let (tr, x, mut r) = (
   std::slice::from_raw_parts(tr_p, sz),
   std::slice::from_raw_parts(x_p, sz),
   std::slice::from_raw_parts_mut(r_p,sz));
 truncate(&tr, &x, &mut r);
}
fn harness() {
 let sz: usize = kani::any_where(|v|
       *v <= 100,000,000);
 let e: usize = kani::any();
 let (mut tr, mut x, mut r) = (vec![e;sz],
     vec![e;sz], vec![e;sz]);
 unsafe {ffi_truncate(sz, &mut tr as *mut _
      as _, &mut x as *mut _ as _, &mut r
     as *mut _ as _);}
}
```

Fig. 8. Buggy function `truncate` and its harness.

Case Study 1. HD+LS. Fig. 8 depicts an arithmetic overflow bug discovered in the function `truncate`, callable from the public FFI `ffi_truncate`. In the `ffi_truncate` harness, the function's arguments are assigned intervals, with the second representing a slice with a length that can vary from 0 to 100 million. The bug occurs when the slice has a size of zero, triggering an arithmetic overflow.

This bug was blocked behind `truncate`'s initial loop, which had an excessively large bound to practically unwind. Nevertheless, by stubbing the initial loop, Kani managed to perform a more thorough analysis that extended beyond the loop's limits, eventually uncovering the arithmetic overflow.

Case Study 2. HD+IS+LS. Fig. 9 shows an out-of-bounds access bug in the unsafe block of `cvt_repr`. This function can be called by the public function `bit_extr`. We created a harness for `bit_extr` as depicted in the figure.

The functions `and`, `xor`, and `xor_assign` are both complex and frequently invoked from within `bit_extr`. Initially, loop stubbing (LS) was applied to the loop, but Kani still got stuck before it could analyze the buggy function `cvt_repr`. The complexity arose from multiple calls to `xor_assign` preceding the invocation of `cvt_repr`, overwhelming the solver. By applying Inter-Proc Scheduling (IS) to `xor`, `xor_assign`, and `and`, Kani successfully reached the analysis of `cvt_repr`. The upper bounds for `nr` and `rb` are 64 and 1, respectively, which places `ix` in the range [0, 64). Meanwhile, the length of `rx` is 1. The out-of-bounds access occurs when `ix` exceeds the length of `rx`.

4.4 Insights and Lessons Learned with Suggestions

We share lessons learned and insights gained from verifying TECC, while also suggesting ways to further improve model-checking tools like Kani for Rust.

```
pub fn bit_extr(
    x: &[usize],
    bit: usize,
    r: &mut [usize]) {
 let r_sz=r.iter().product();//[8,64]
 let x_sz=x.iter().product();//[1,1]
 for i in 0..bit { // loop bound: 32
    xor(...);
    and(...);
    xor_assign(...);
  }
  xor_assign(..);
  xor_assign(..);
  xor_assign(..);
  xor_assign(..);
  let (rx, es) = ([x_sz], [r_sz]);
  cvt_repr(&rx, &es);
}
```

```
fn cvt_repr(rx:&[usize], es:&[usize]){
 let dm = rx.len()-1;
 let rb = rx[dm]; // rb: [1, 1]
 let nr = es.iter().take(rb).product();
 for nd in 0..nr { // nd: [0, 64)
  for i in 0..rb { // i: [0, 1)
   unsafe {
    let ix = nd * rb; // ix: [0, 64)
    let bx = rx.get_unchecked(ix);//Acc-
     ess out-of-bound!
    //...
}
fn harness() {
 let bt=kani::any_where(|v|*v<=32);
 let sz=kani::any_where(|v|*v<=2&&*v>=1);
 let (x, mut r)=(vec![1;sz],vec![8;sz]);
 bit_extr(&x, bit, &mut r);
}
```

Fig. 9. Buggy function `cvt_repr` and its harness.

Verification Scope. In Rust projects, unsafe code constitutes a small portion of the codebase [3]. To ensure a balance between effort and effectiveness, it is important to define the appropriate scope for verification. In practice, verifying public functions that can reach unsafe code is sufficient since unsafe code is often executed through exposed functions from other crates.

Harness Development. Generating high-quality harnesses is paramount for the verification process. Utilizing integration instead of unit tests is advisable, as they offer a more comprehensive understanding of how functions are utilized in real-world scenarios. Collaboration with developers is crucial to establish the appropriate input space for the functions under verification. Comprehensive code coverage serves as a reliable metric for assessing the effectiveness of verification efforts. Additionally, harness development provides an opportunity to review the codebase, allowing for the removal of unused public functions to minimize potential attack surfaces to unsafe code.

Loop Stubbing. Loop stubbing is highly effective for large codebases with complex loops. It addresses challenges that BMC encounters when analyzing intricate loops, facilitating in-depth analysis. The key approach is to maintain memory safety checks within loops while approximating external side effects, ensuring thorough memory safety analysis without compromising soundness.

Function Verification Order and Function Stubbing. In real-world codebases like TECC, managing the verification order of complex and frequently-invoked functions, such as `chebyshev`, can significantly boost BMC efficiency. Our inter-procedural scheduling, which includes function stubbing, cuts TECC's verification time by about threefold compared to the non-stubbing alternative.

False Positives. Although over-approximation in loop and function stubbing typically leads to false positives, our verification of TECC did not produce any false positives during the stubbing process. This is because TECC's loops and functions operate on unrestricted values within their respective domains.

Stubbing Generation. Stubbing in TECC verification applies to both functions and loops. Function stubbing automatically over-approximates side effects using unconstrained values, which is effective given TECC functions operate within specific unconstrained domains. Loop stubbing, which is currently semi-automatic, over-approximates side effects but requires manual effort to ensure completeness of memory safety checks. It could potentially be fully automated. In the context of Rust's MIR being lowered to LLVM IR during compilation, implementing automatic safety checks could involve developing LLVM passes that identify loop side effects, approximate them with unconstrained domain values, and insert memory safety checks before each loop memory access.

Rust-Specific Memory Safety. Ensuring memory safety in a Rust codebase with unsafe code requires verifying both the unsafe and the impacted safe Rust code. For TECC, we verified an additional 4,099 lines of safe Rust, which is 36% more than the unsafe code. Minimizing the use of unsafe code is crucial to simplify the verification process and reduce memory safety risks.

Feedback from Developers. During the TECC verification process, developers valued our formal verification practice, especially for identifying 39 memory-safety issues that were overlooked by their existing static and dynamic analysis tools. They now plan to incorporate this verification process into their daily development routines to enhance memory safety.

Limitations. We employed Kani-0.22.0 [15] for model checking the Rust codebase. Kani provides interfaces for function stubbing, assigns value ranges to symbolic variables, and supports user property assertions. However, it could benefit from incorporating function contracts and loop invariants. The `kani::stub()` function has limitations, especially with trait methods. Additionally, Kani does not yet support all documented undefined behaviors. Moreover, when Kani stalls during loop analysis, providing output that specifies the loop and progress would aid users in identifying where stubbing is necessary.

5 Conclusion

We introduce *UnsafeCop*, an approach for ensuring memory safety in real-world Rust projects with unsafe Rust code. *UnsafeCop* identifies functions exposed to other crates that execute unsafe code, transforms tests into harnesses, determines loop bounds using abstract interpretation, and applies loop stubbing for large-bounded loops with memory-safety checks. For bounded model checking, we employ Kani, optimizing the verification order of functions and incorporating function stubbing to enhance performance. We tested our approach on TECC, a real-world project combining trusted computing with secure multi-party computation, consisting of 30,174 lines of Rust code, of which 3,019 are unsafe.

UnsafeCop identified and verified 39 memory safety issues, reducing verification time by 73.71% compared to the non-stubbing alternative. These results demonstrate the effectiveness of *UnsafeCop* in improving memory safety in Rust projects.

References

1. Ardito, L., et al.: Rust-code-analysis: a rust library to analyze and extract maintainability information from source codes. SoftwareX **12**, 100635 (2020). https://doi.org/10.1016/j.softx.2020.100635
2. Astrauskas, V., et al.: The prusti project: formal verification for rust. In: NASA Formal Methods Symposium, pp. 88–108. Springer (2022). https://doi.org/10.1007/978-3-031-06773-0_5
3. Astrauskas, V., Matheja, C., Poli, F., Müller, P., Summers, A.J.: How do programmers use unsafe rust? Proceedings of the ACM on programming languages **4**, 1 – 27 (2020). https://api.semanticscholar.org/CorpusID:220859132
4. Biere, A., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2021-1, pp. 10–13. University of Helsinki (2021)
5. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the 2nd International Symposium on Programming, Paris, France, pp. 106–130. Dunod (1976)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252 (1977)
7. Cui, M., Chen, C., Xu, H., Zhou, Y.: Safedrop: detecting memory deallocation bugs of rust programs via static data-flow analysis. ACM Trans. Softw. Eng. Methodol. **32**(4) (2023). https://doi.org/10.1145/3542948
8. Denis, X., Jourdan, J.H., Marché, C.: Creusot: a foundry for the deductive verification of rust programs. In: International Conference on Formal Engineering Methods, pp. 90–105. Springer (2022). https://doi.org/10.1007/978-3-031-17244-1_6
9. Ebert, C., Cain, J., Antoniol, G., Counsell, S., Laplante, P.: Cyclomatic complexity. IEEE Softw. **33**(6), 27–29 (2016)
10. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: International Conference on Computer Aided Verification, pp. 343–361. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20
11. Halstead, M.H.: Elements of Software Science (Operating and programming systems series), Elsevier Science Inc. (1977)
12. Harmonic mean. https://en.wikipedia.org/wiki/Harmonic_mean
13. Höltervennhoff, S., Klostermeyer, P., Wöhler, N., Acar, Y., Fahl, S.: {"I} wouldn't want my unsafe code to run my {pacemaker"}: an interview study on the use, comprehension, and perceived risks of unsafe rust. In: 32nd USENIX Security Symposium (USENIX Security 23), pp. 2509–2525 (2023)
14. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL), 1–34 (2017)

15. Kani rust verifier. https://github.com/model-checking/kani
16. Kroening, D., Tautschnig, M.: CBMC–C bounded model checker: (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20, pp. 389–391. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
17. Li, Z., Wang, J., Sun, M., Lui, J.C.: Mirchecker: detecting bugs in rust programs via static analysis. In: Proceedings of the 2021 ACM SIGSAC conference on computer and communications security, pp. 2183–2196 (2021)
18. Li, Z., Wang, J., Sun, M., Lui, J.C.: Detecting cross-language memory management issues in rust. In: European Symposium on Research in Computer Security. pp. 680–700. Springer (2022). https://doi.org/10.1007/978-3-031-17143-7_33
19. Lindner, M., Aparicius, J., Lindgren, P.: No panic! verification of rust programs by symbolic execution. In: 2018 IEEE 16th International Conference on Industrial Informatics (INDIN, pp. 108–114. IEEE (2018)
20. Matsushita, Y., Denis, X., Jourdan, J.H., Dreyer, D.: Rusthornbelt: a semantic foundation for functional verification of rust programs with unsafe code. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 841–856 (2022)
21. Matsushita, Y., Tsukada, T., Kobayashi, N.: Rusthorn: CHC-based verification for rust programs. ACM Trans. Program. Lang. Syst. (TOPLAS) **43**(4), 1–54 (2021)
22. Mirai: Rust mid-level IR abstract interpreter. https://github.com/facebookexperimental/MIRA
23. Miri: an interpreter for rust's mid-level intermediate representation. https://github.com/rust-lang/miri
24. Qin, B., Chen, Y., Yu, Z., Song, L., Zhang, Y.: Understanding memory and thread safety practices and issues in real-world rust programs. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 763–779 (2020)
25. Rakamarić, Z., Emmi, M.: Smack: decoupling source language details from verifier implementations. In: Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26, pp. 106–113. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_7
26. Rust verification tools. https://project-oak.github.io/rust-verification-tools/about.html
27. Shen, Y., et al.: Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 955–970 (2020)
28. Tomb, A.: Crux: Introducing our new open-source tool for software verification (2020)
29. Trustinsoft analyzer. https://trust-in-soft.com
30. Ullrich, S.: Simple verification of rust programs via functional purification, Master's Thesis, Karlsruher Institut fr Technologie (KIT) (2016)
31. Unsafe superpowers. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

32. Wang, H., et al.: Towards memory safe enclave programming with rust-sgx. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 2333–2350 (2019)
33. Xu, H., Chen, Z., Sun, M., Zhou, Y., Lyu, M.: Memory-safety challenge considered solved. Empirical Study Rust CVEs. CoRR, abs/2003.03296 (2020)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Beyond the Bottleneck: Enhancing High-Concurrency Systems with Lock Tuning

Juntao Ji[1], Yinyou Gu[1], Yubao Fu[1,2], and Qingshan Lin[1,2](✉)

[1] Alibaba Cloud Computing Co. Ltd., Beijing, China
{juntao.jjt,yinyou.gyy,yubao.fyb,qingshan.lqs}@alibaba-inc.com
[2] Apache RocketMQ Community, New York, USA
{fuyou,linhill}@apache.org

Abstract. High-concurrency systems often suffer from performance bottlenecks [1]. This is often caused by waiting and context switching caused by fierce competition between threads for locks. As a cloud computing company, we place great emphasis on maximizing performance. In this regard, we transform the lightweight spin-lock and propose a concise parameter fine-tuning strategy, which can break through the system performance bottleneck with minimal risk conditions. This strategy has been validated in Apache RocketMQ [2], a high-throughput message queue system. Through this strategy, we achieved performance improvements of up to 37.58% on X86 CPU and up to 32.82% on ARM CPU. Furthermore, we confirmed the method's consistent effectiveness across different code versions and IO flush strategies, demonstrating its broad applicability in real-world settings. This work offers not only a practical tool for addressing performance issues in high-concurrency systems but also highlights the practical value of formal techniques in solving engineering problems.

Keywords: High-Concurrency systems · Performance optimization · Spin-lock backoff

1 Introduction

In recent decades, computer processors have undergone significant transformations. Initially, performance enhancements were driven by increasing the clock frequency of single-core processors and transistor density. However, this trend was rendered unsustainable due to energy consumption and heat dissipation. As a solution, the industry shifted towards multicore processors, aligned with Moore's Law [4], which predicts continued transistor density doubling. Today, multicore processors are equipped in personal computers, smartphones, gaming consoles, and even servers and supercomputers. This allows parallel processing of more tasks.

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 325–337, 2025.
https://doi.org/10.1007/978-3-031-71177-0_20

Multicore processors, however, do not improve software performance linearly. On the contrary, unlocking the parallel potential of these cores presents a significant emerging challenge [1]. Multicore environments require software to effectively distribute tasks across multiple cores, involving complex synchronization and coordination issues. Without proper task allocation and data access management, errors such as data races and deadlocks can occur during execution. These errors are not present in sequentially executed programs [6]. Moreover, due to the interdependencies among cores in multicore processors, parallel solutions to single problems must be carefully designed to avoid performance bottlenecks and resource waste.

Concurrent programming is essential because it could unlock multicore processors' full capabilities, enabling complex tasks and large volumes of data [9]. Concurrent programming also challenges system performance and stability. Over years of development, a highly concurrent system may evolve into an exceedingly complex structure. This is often accompanied by complex data flows and performance constraints among system components. While locking mechanisms can ensure concurrent operations' safety, if not used properly, they may become performance bottlenecks themselves. Over time, these issues-if not carefully managed-can lead to performance degradation, affecting response time and reliability.

Locks serve as a fundamental mechanism for synchronizing multiple execution threads for safe access to shared resources. Although locks are widely used in multicore embedded systems to ensure mutual exclusion, their usage also presents challenges. As concurrent systems become increasingly complex, proper lock management strategies are crucial for maintaining system performance [7]. Researchers have been studying and developing various lock implementations to optimize performance, fairness, and predictability for different scenarios. However, in practice, the choice of the appropriate lock type for a specific scenario is often based on developers' assumptions, which may not match actual requirements. Incorrect lock types can degrade performance and result in unfair resource access.

Research has found that no single lock implementation performs best in all scenarios [13], and some well-performing locks may exhibit severe performance issues in other contexts. These findings underscore the necessity of lock performance analysis, such as the ability to monitor lock contention intensity in real-time and adjust lock strategies accordingly [13]. However, real-time monitoring of lock contention in a process can incur a performance overhead, and such adjustments may be reactive rather than proactive.

Therefore, we do not seek to find a "one-size-fits-all" lock-the cost is simply too prohibitive. Our goal is to discover a straightforward approach that allows us to fine-tune locks in a simple manner, enabling them to unlock the performance potential of specific applications. With this in mind, the lock overhead must be kept to a minimum, and it's imperative that we autonomously manage the CPU utilization strategy of the lock to overcome current performance bottlenecks. To this end, we propose a method for optimizing spin-lock parameters. This method, based on an M/G/1 queue model [5] analysis of lock overhead, aims to identify the optimal spin backoff parameter k in high-pressure scenarios to balance the costs associated with lock acquisition. By following this method, applications in high-concurrency scenarios can perform better.

With this theory, we adjust the spinning behavior of locks within the system to flexibly decide on the spin threshold between different scenarios. This allows us to find the best performance balance while maintaining system stability. This optimization technique has been validated in the open-source Apache RocketMQ as well as in commercial RocketMQ instances offered by Alibaba Cloud. This has resulted in a significant improvement in message sending performance by 37.58% on normal X86 CPU.

Furthermore, to validate the universality of this optimization method, we deployed RocketMQ on servers equipped with Alibaba Cloud's newly developed ARM architecture CPU. We applied the optimization strategy described in this study. The optimized system performance achieved an additional 32.82% improvement, proving our method's effectiveness is not limited by a specific hardware architecture. We also tested the performance of this strategy when deploying older code versions on different CPUs or when employing various data persistence strategies. Ultimately, all scenarios resulted in performance improvements. These performances confirm that our method can improve the overall performance of high-concurrency systems dealing with chaotic data flows and inter-component performance constraints.

2 Preliminaries

2.1 Apache RocketMQ

Apache RocketMQ [2] is a cloud-native distributed messaging and streaming platform designed for real-time data processing that spans collaboration scenarios across cloud, edge, and devices. Initially created by Alibaba Group, it was donated to the Apache Software Foundation in 2016 and has since become a prominent top-level project within the foundation. RocketMQ excels in handling diverse message queue models, including publish/subscribe, point-to-point messaging, delayed message delivery, and message sequencing. These capabilities fulfill the stringent requirements of applications that demand high scalability, reliability, and throughput.

RocketMQ's adoption across various industries such as finance, e-commerce, Internet of Things (IoT), and big data analysis is a testament to its adaptability and capability to meet complex messaging challenges. The architecture of RocketMQ is centered around clusters of Brokers and NameServers. The Brokers manage message storage, lifecycle, and distribution, whereas NameServers play an essential role in service discovery and message routing with their lightweight design.

Additionally, RocketMQ supports transactional messaging and provides client libraries for popular programming languages such as Java, C++, and Go, making it easier for developers to build and scale high-performance distributed applications.

Known for its impressive performance, RocketMQ can process several hundred thousand messages per second without compromising stability or reliability. The system's design inherently favors distributed deployment, which allows for effortless scalability to meet growing business needs. The platform also includes an extensive set of monitoring metrics and tools, simplifying management and operational tasks.

Our team, the original developers of Apache RocketMQ, has dedicated considerable effort to advance its performance, focusing on maximizing throughput per machine. This commitment to performance enhancement is a driving force behind the research presented in this paper, situating our work within the broader context of innovation in distributed messaging systems.

2.2 Spin-Lock

For decades, spin-locks have been a key study subject in concurrent programming. They are lauded for their role in regulating access to shared resources in multicore systems. Recognized for their minimal overhead in low-contention scenarios, spin-locks shine where locking time is expected to be short, allowing threads to effectively 'spin' until a resource becomes available [3].

The simplest and perhaps most rudimentary spin-lock form is the test-and-set (TAS) lock. This implementation employs a brute-force approach, spinning aggressively using atomic operations to gain exclusive access to a resource. Although this simplicity is compelling, it's also its Achilles' heel: atomic operations can aggressively drain CPU execution cycles and negatively impact shared resources like the system's bus and memory, thus hampering the performance of the spinning thread and reducing overall system throughput, more so when multiple threads contend for the same lock [3].

To address the performance challenges associated with TAS, researchers proposed the test-test-and-set (TTAS) lock, an iteration aiming to reduce the heavy use of atomic operations [10], TTAS lock attempts to ameliorate the spin-lock behavior by integrating a preliminary, non-atomic check that the lock is potentially free before invoking atomic operations. This strategy reduces unnecessary bus traffic and cache coherence invalidations when the lock is known to be held. Nonetheless, TTAS still struggles with performance under high contention, which is exacerbated in cache-coherent environments where threads vying for a lock can cause cacheline invalidation due to coherence traffic-leading to additional delays and memory access overhead [13].

To tackle the shortcomings of both TAS and TTAS in high-contention scenarios, we propose a spin-backoff strategy. Rather than spinning indiscriminately, we employ a strategy where contending threads back off after a certain number of collisions. This mitigates collision risk and, as a result, enhances system performance. This strategy has proven to enhance complex systems' performance under high concurrency.

3 Modeling Spin-Lock Overheads

3.1 Fundamental Assumptions

To effectively model the problem, we consider an exponential distribution model of lock contention probability and an M/G/1 queue model [5]. The M/G/1 queueing model is a type of queueing theory model where arrivals follow a Poisson process (denoted by M), service times have a general distribution (denoted by G), and there is a single server (denoted by 1). This model is used to analyze and describe characteristics such as customer waiting times and queue lengths in single-server systems. This necessitates the redefinition of lock behavior and system load. We consider several key variables in our model:

- Arrival rate (λ): The average rate of lock requests per unit time.
- Service rate (μ): The average rate at which locks are released and successfully acquired by another thread per unit time.
- System utilization ratio ($\rho = \lambda/\mu$): The ratio of the arrival rate to the service rate.

In the normal M/G/1 queue model, the arrival process is a Poisson process, and the service time distribution is a general distribution. When simulating the waiting time for a spin-lock, assuming that the lock holding time follows an exponential distribution means that each thread attempting to acquire the lock has an equal chance of success at any moment, unaffected by the previous waiting time. Therefore, in our case, we assume the lock holding time follows an exponential distribution, simplifying the model to an M/M/1 queue model because the service time is also exponentially distributed.

One of the critical properties of the M/M/1 queue model is the average queue length (L_q), which is given by the formula:

$$L_q = \frac{\rho^2}{1-\rho}$$

This formula shows how the average queue length can be directly determined by system utilization (ρ), and is independent of service time variance due to the nature of the exponential distribution. Then the average waiting time in the queue W_q is given by the formula:

$$W_q = \frac{\rho}{\lambda(1-\rho)}$$

3.2 Modeling Process

To tailor our spin-lock model, we introduce additional parameters:

- Spin time (T_s): The average time for each spin attempt, which equates to a single CAS (Compare-And-Swap) operation, which is typically on the order of nanoseconds.

- Context switch time (T_c): The average time for each context switch, encompassing operations such as saving the current process state, loading another process's state, and potential cache invalidation, with the time overhead generally in the order of microseconds.
- Number of spins (k): The number of attempts a thread makes to acquire the lock before yielding.

Assuming a thread acquires the lock on the ith attempt ($1 \leq i \leq k$), the expected spin time is $i \cdot T_s$. With each spin attempt being independent, and the probability of acquiring the lock on any attempt being $1 - \rho$, the probability of success on the ith attempt is $\rho^{i-1} \cdot (1 - \rho)$.

We consider the total expected time $E(T_{\text{total}})$ as the weighted sum of the spin attempts, where the weights are the probabilities of success for each attempt. The expected spin time $E(T_{\text{spin}})$ is expressed as:

$$E(T_{\text{spin}}) = \sum_{i=1}^{k} (i \cdot T_s) \cdot \rho^{i-1} \cdot (1 - \rho)$$

Defining $H(x) = \sum_{i=1}^{k} i \cdot x^{i-1}$, we can simplify the $E(T_{\text{spin}})$ to:

$$E(T_{\text{spin}}) = T_s \cdot (1 - \rho) \cdot H(\rho)$$

To facilitate the calculation of $E(T_{\text{spin}})$, we employ a summation technique involving geometric series and its derivatives. Let $G(x) = \sum_{i=0}^{k-1} x^i$ represent our geometric series, with the sum $G(x) = \frac{1-x^k}{1-x}$. We note that $H(x)$ is the derivative of $G(x)$, resulting in:

$$H(x) = G'(x) = \frac{(1 - x^k) - kx^{k-1}(1 - x)}{(1 - x)^2}$$

By substituting x with ρ, we obtain $H(\rho)$, which is then used to compute $E(T_{\text{spin}})$:

$$E(T_{\text{spin}}) = T_s \cdot (1 - \rho) \cdot H(\rho) = T_s \cdot \frac{(1 - \rho^k) - k\rho^{k-1}(1 - \rho)}{1 - \rho}$$

If a thread fails to obtain the lock after k spins, it performs a context switch and subsequently waits in the queue. The expected time for this event is:

$$E(T_{\text{yield-total}}) = \rho^k \cdot (k \cdot T_s + T_c + W_q)$$

The total expected waiting time $E(T_{\text{total}})$ is thus the sum of the expected spin time and the expected yield time:

$$E(T_{\text{total}}) = E(T_{\text{spin}}) + E(T_{\text{yield-total}})$$

$$E(T_{\text{total}}) = T_s \cdot \frac{(1 - \rho^k) - k\rho^{k-1}(1 - \rho)}{1 - \rho} + \rho^k \cdot \left(k \cdot T_s + T_c + \frac{\rho}{\lambda(1 - \rho)}\right)$$

3.3 Validation

To validate our model, we examine the behavior of the expected total time as system utilization (ρ) approaches different limits. For a system under minimal load, where ρ approaches 0, the expected spin time simplifies to:

$$\begin{aligned}\lim_{\rho\to 0} E(T_{\text{total}}) &= \lim_{\rho\to 0} T_s \cdot \frac{(1-\rho^k) - k\rho^{k-1}(1-\rho)}{1-\rho} \\ &\quad + \lim_{\rho\to 0} \rho^k \cdot \left(k \cdot T_s + T_c + \frac{\rho}{\lambda(1-\rho)}\right) \\ &= T_s + 0 \\ &= T_s\end{aligned}$$

This implies that the spin cost is equivalent to T_s, meaning that lock acquisition occurs immediately after a single compare-and-swap (CAS) operation.

Conversely, as the system load approaches its maximum capacity and ρ approaches 1, the expected spin success time becomes indeterminate and requires the application of l'Hôpital's rule:

$$\lim_{\rho\to 1} E(T_{\text{spin}}) = \lim_{\rho\to 1} T_s \cdot \frac{(1-\rho^k) - k\rho^{k-1}(1-\rho)}{1-\rho}$$

$$\lim_{\rho\to 1} E(T_{\text{spin}}) = \lim_{\rho\to 1} \frac{-k - k(k-1) + k^2}{-1} = 0$$

This result indicates that the expected spin time to acquire the lock is zero. Thus, the predominant component of the total expected time at high system utilization is:

$$\lim_{\rho\to 1} E(T_{\text{total}}) = \lim_{\rho\to 1} \rho^k \cdot \left(k \cdot T_s + T_c + \frac{\rho}{\lambda(1-\rho)}\right)$$

Hence, in scenarios where ρ is near 1, the time cost comprises k spins, a context switch, and the waiting time in the queue.

In summary, when ρ is low, increasing k rapidly reduces the probability of a thread spinning k times without acquiring the lock, minimizing context switch overhead. Conversely, when ρ is high, the main contribution to waiting time will be $E(T_{\text{yield-total}})$, which necessitates careful control of k. A low k leads to more frequent context switching, while a high k results in extended spin times, reducing lock throughput. This highlights the need for an optimal spinning strategy that balances $k \cdot T_s$ against $T_c + \frac{\rho}{\lambda(1-\rho)}$ when ρ further increases, leading to longer queueing times.

4 Spin-Lock Fine-Tuning

Following the mathematical foundation laid down in the preceding section, our focus shifts toward the practical application of the model we have established.

Our primary goal is to strategically determine the optimal value of k, the number of spin attempts. This minimizes the total expected waiting time for a thread contending for a lock within a system operating under varying loads, represented by ρ.

System load, ρ, fundamentally affects spin-lock performance. At peak system loads, we expect lock contention to be at its highest. This translates into a higher probability that a thread will have to wait before acquiring the lock. This scenario is particularly pertinent for optimizing our spin-lock strategy, as lock contention costs increase.

4.1 Strategy Overview

1. **Peak Load Simulation**: The first step involves pushing the system to its maximum load to simulate an environment where lock contention is at its highest. By doing so, we ensure that ρ reflects a state of maximum contention, and that any optimizations we perform are directed at the most stressful operating conditions.
2. **Dynamic Tuning of k**: In this high contention scenario, we begin with a minimum k value of 1 and incrementally adjust it while monitoring the impact on system performance metrics. Considering that ρ remains relatively constant under peak load, our task consists of determining the optimal value of k. This balances the trade-off between spinning costs and context switching costs and potential delays.
3. **Performance Optimization and Monitoring**: We aim to find the optimal k value that boosts the lock's performance. When increasing k further reduces performance gains, it's no longer beneficial to keep spinning. This ideal k ensures our spin-lock operates at peak efficiency under the existing system load, which means we have identified the best balance between $k \cdot T_s$ and $T_c + \frac{\rho}{\lambda(1-\rho)}$.
4. **Mutex Adoption Strategy**: If performance degrades as k increases from its initial value, it suggests that the lock contention is too intense for spinning to be effective. In such scenarios, a Mutex lock, which involves less spinning, may be more appropriate. This decision is informed by the understanding that, under extreme contention, excessive spinning's overhead outweighs its benefits.

The strategy we propose provides a systematic approach to identifying an optimal k value under peak system load. This value aims to balance the probability of lock acquisition against the costs associated with prolonged spinning and context switching. Importantly, while the optimal k is derived under high contention conditions, it offers a benchmark that performs efficiently across a range of loads. When the system load diminishes, the optimal k remains effective at quickly acquiring locks while minimizing context switching overhead.

In essence, this strategy does not limit its applicability to a single ρ but instead offers a versatile solution that accommodates the entire spectrum of system loads. By integrating our theoretical insights with practical, empirical observations, we ensure that our spin-lock strategy is both robust and adaptive. This is capable of maintaining lock performance in the face of fluctuating system demands.

5 Experiment

5.1 Variables

To demonstrate the universality of our spin-backoff strategy, we designed multiple scenarios for testing. The results indicate that our strategy significantly improves Apache RocketMQ performance in various contexts. The scenario variables are as follows:

Different CPU Architectures: Throughout its development journey, Apache RocketMQ, initially designed and built for the x86 architecture, has successfully been ported and now supports running on ARM architecture CPUs. This adapts to hardware architecture diversification. Running on various CPU architectures challenges RocketMQ's code development. Therefore, we tested our proposed optimization strategy on more than one type of CPU to prove its effectiveness across multiple architectures. In addition to the traditional x86 CPU, we tested Alibaba Cloud's self-developed ARM CPUs-a novel architectural choice, indicating that Apache RocketMQ was not previously optimized for this architecture.

Different Code Versions: Over approximately a decade of iterations, Apache RocketMQ has undergone significant changes-adding many enhanced features and increasing the complexity of message publishing and receiving processes. Thus, we validated not only with the latest code but also with a stable version from two years ago. This was to prove the enduring effectiveness of our strategy throughout code evolution.

Different Flush Policies: Apache RocketMQ is a message queue with built-in storage, so bottlenecks may arise from more than CPU performance. We also set different flush policies to simulate various data persistence approaches. An aggressive persistence approach (ASYNC) leads to asynchronous flushing, offering higher throughput at the risk of data loss in a crash. A conservative persistence approach (SYNC) uses synchronous flushing, which doesn't report success to producers until messages are successfully written to disk. This ensures data integrity at the expense of lower throughput.

5.2 Experimental Procedure

Based on the aforementioned variables, we arranged combinations for a total of $2^3 = 8$ scenarios of maximum original throughput. We applied our optimization strategy to each. Ultimately, our strategy identified an optimal spin parameter k in every high-pressure scenario, enhancing maximum original performance.

With the three scenarios serving as variables, we kept all other related parameters constant:

A 16 vCPU, 32 GiB Alibaba Cloud model with CentOS 7.9 64-bit, ESSD cloud PL1 disk (1024 GiB, 50000 IOPS), an internal network bandwidth of 10 Gbit/s, and a network packet transmission rate of 3 million PPS were selected.

For the experiments in this chapter, we employed multiple stress test machines to send messages at full capacity to Apache RocketMQ servers. To mitigate disk pressure caused by high message volume, we set the message payload to just 2 bytes. This makes the total message size approximately 100 bytes. During this process, we implemented a backpressure strategy that kept the Broker's processing load close to but not exceeding its limits. This is consistent with our theoretical design where ρ approaches 1.

Under these conditions, we tested the raw limit of performance (Send QPS) and applied the spin parameter optimization strategy mentioned in this paper. We obtained the optimal spin parameter k along with performance post-optimization. Table 1 records these metrics and calculates the final percentage of performance optimization.

Table 1. Performance Improvement Overview

Code Version	CPU Arch	Flush Policy	Original QPS	k	Optimal QPS	Improvement
2 years ago Code	X86	ASYNC	165516.84	10^2	208376.04	+25.89%
		SYNC	109614.92	10^2	125551.24	+14.54%
	ARM	ASYNC	144412.04	10^0	191813.80	+32.82%
		SYNC	119083.36	10^2	125677.16	+5.45%
New version Code	X86	ASYNC	111090.20	10^3	150968.20	+35.90%
		SYNC	112671.60	10^3	155019.20	+37.58%
	ARM	ASYNC	168707.52	10^3	191520.40	+13.52%
		SYNC	175771.76	10^3	194051.32	+10.40%

Table 1 exemplifies that, under all variable scenarios, our most effective spin parameter search strategy is effective-by optimizing a single spin-lock during message sending, we can achieve a performance boost ranging from 5.45% to 37.58%.

The strategy shines especially on ARM CPUs. Taking the aggressive flush policy under the ARM architecture as an example, after two years of code iteration, Apache RocketMQ has improved message sending throughput by 24,000 QPS, approximately 16.8%. If the proposed strategy had been implemented in the code version two years ago, it would have directly improved performance by 32.82%. This is twice the improvement achieved after two years of iterations.

Obtaining such a significant performance boost in traditional high-concurrency systems is typically difficult, as it often implies code refactoring and phased rollout risks. Nevertheless, optimizing the spin-lock backoff strategy brings risk-free performance gains while maintaining thread safety under high concurrency. The improvement is akin to infusing water into a cup filled with sand-better utilizing CPU resources otherwise wasted by spinning behavior.

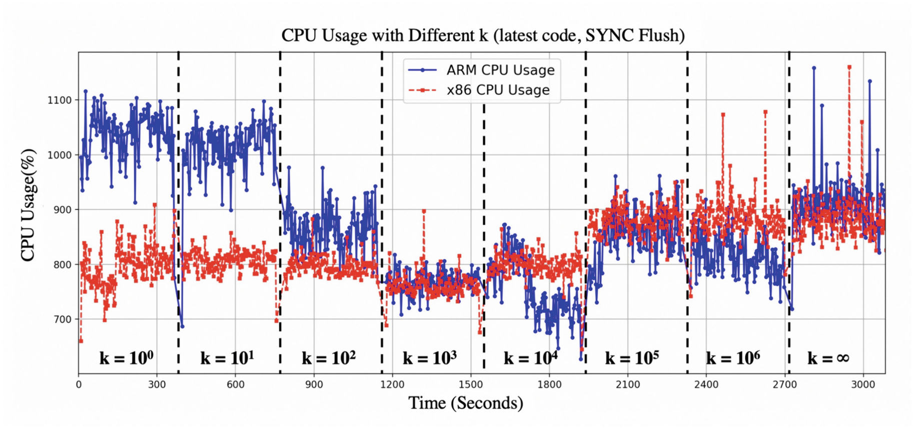


Fig. 1. CPU usage with different k in the new version code, and SYNC flush policy.

In addition, we also examined the CPU usage across various values of k. This usage was measured based on the percentage CPU utilization ("CPU%") reported by the **pidstat** command. Each experimental set was conducted for a duration of six minutes and a total of eight sets of k values were tested. The experiments tested k ranging from 10^0 to 10^6, as well as the scenario where k was set to infinity (indicating continuous spinning). We take the most remarkable set of results as an example, specifically the latest code version deployed on a machine with an X86 CPU, utilizing the SYNC flush mode. The results are presented in the Fig. 1.

The experimental results reveal that at $k = 10^3$, not only did our sending speed reach its peak (155019.20), but the CPU usage was also at its lowest. This suggests that our spin-backoff strategy successfully conserved CPU resources, thereby enhancing CPU utilization. At this point, the CPU supported higher performance levels with lower utilization rates, indicating that the performance bottleneck had shifted-perhaps to the disk, for example.

In Table 1, we observed a 10.4% improvement in the performance of RocketMQ on an ARM CPU with the same k (10^3) and configuration parameters (latest code, SYNC Flush mode). Furthermore, as illustrated in Fig. 1, CPU usage experienced a substantial decrease when $k = 10^3$, falling from an average of over 1000% to around 750%. This decrease in resource consumption indicates that alleviating other system bottlenecks could lead to even more significant gains in performance.

6 Conclusion & Future Work

In this paper, we focus on a common and challenging issue in high-concurrency systems: performance bottlenecks. This paper introduces an innovative approach utilizing a back-off spin-lock strategy to tackle performance bottlenecks. A cost analysis of spin-locks was conducted to establish a quantitative relationship linking the expected overhead of lock contention with the number of spin attempts (k) and system load (ρ). On the foundation of theoretical modeling, we explored a viable solution: tuning parameters at peak system loads, where ρ is close to 1, to find the optimal balance between spin wait and context switch times. This approach has significantly improved system performance. Moreover, at lower ρ values, the determined maximal spin attempts (k) naturally become ineffective, while also reducing context switching costs, thus ensuring operational efficiency under light load conditions.

The strategy for searching for the backoff spin-lock parameter, as proposed in this article, has been empirically validated on Apache RocketMQ as well as in commercial RocketMQ instances offered by Alibaba Cloud. Our tests confirmed the strategy's effectiveness across different CPU architectures, including X86 and ARM, demonstrating its broad applicability. Moreover, we have evaluated the strategy's stability by examining its performance across various versions of Apache RocketMQ, observing consistent and reliable behavior over time. The study also explored how the strategy performs in conjunction with different data flush approaches-both asynchronous and synchronous-ensuring that the backoff spin-lock optimization effectively enhances system performance under diverse system behaviors. In our experiments, the application of this spin-lock backoff parameter search strategy led to performance improvements ranging from 5.45% to 37.58%, showcasing its potential for enhancing system efficiency in multiple contexts.

The performance enhancements reported in this paper are invigorating within the industry - minute optimizations are hard-won in complex high-concurrency systems, let alone improvements as substantial as 30% at their peak. This finely tuned optimization method avoids extensive code overhauls or phased rollouts. However, it brings safe and stable improvements to high-concurrency systems. It is like strategically pouring water into a cup of densely packed sand. This utilizes CPU resources otherwise wasted on spinning and context switching.

Looking ahead, we aim to continue our exploration in both laboratory settings and industrial arenas to discover methods to quantify system load (ρ) accurately. This insight will provide us with additional controllable parameters, helping us refine our spin-lock strategy further. Consequently, it will provide more accurate and efficient performance optimization solutions for various high-concurrency systems. Through these endeavors, we aspire to contribute to deeper technological advancements and breakthrough industrial applications within high-concurrency systems.

References

1. Han, S., Dang, Y., Ge, S., Zhang, D.: Performance debugging in the large via mining millions of stack traces. In: Proceedings of the International Conference on Software Engineering, pp. 176–186 (2012)
2. Apache RocketMQ. Apache Software Foundation. Available online: http://rocketmq.apache.org (Accessed on 1 Apr 2024)
3. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2012). ISBN 978-0-12-397337-5
4. Moore, G.E.: Progress in digital integrated electronics. In: Proceedings of the IEEE Electron Devices Meeting, vol 21, pp. 11–13, San Francisco, CA (1975)
5. Nelson, R.: The M/G/1 queue. In: Probability, Stochastic Processes, and Queueing Theory. Springer, New York, NY (1995). https://doi.org/10.1007/978-1-4757-2426-4_7
6. Breshears, C.: The Art of Concurrency: A ThreadMonkey's Guide to Writing Parallel Applications. O'Reilly Media, Sebastopol, CA, USA (2009)
7. Anderson, T.E.: The performance of spin lock alternatives for shared-money multiprocessors. IEEE Trans. Parallel Distrib. Syst. **1**(1), 6–16 (1990). https://doi.org/10.1109/71.80120
8. Wikipedia contributors. *M/G/1 Queue*. https://en.wikipedia.org/wiki/M/G/1_queue. April 2024. [Accessed 10 Apr 2024]
9. Akhter, S., Roberts, J.: Multi-Core Programming: Increasing Performance through Software Multi-Threading, vol. 33 (2006). Hillsboro, OR, USA: Intel Press
10. Anderson, T., Dahlin, M.: Operating systems: Principles and Practice, 2nd ed. Recursive Books (2014). ISBN 978-0-9856735-2-9
11. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**(1), 21–65 (1991). https://doi.org/10.1145/103727.103729
12. Dice, D., Shavit, N.: What really makes transactions faster? (2006)
13. Li, L., Wagner, P., Mayer, A., Wild, T., Herkersdorf, A.: A non-intrusive, operating system independent spinlock profiler for embedded multicore systems. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 322–325 (2017). https://doi.org/10.23919/DATE.2017.7927009

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

AGVTS: Automated Generation and Verification of Temporal Specifications for Aeronautics SCADE Models

Hanfeng Wang[1], Zhibin Yang[1](✉), Yong Zhou[1], Xilong Wang[1], Weilin Deng[1], and Wei Li[2]

[1] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
yangzhibin168@163.com
[2] Aerospace Life-support Industries Ltd., Xiangyang, China

Abstract. SCADE is both a formal language and a model-based development environment, widely used to build and verify the models of safety-critical system (SCS). The SCADE Design Verifier (DV) provides SAT-based verification. However, DV cannot adequately express complex temporal specifications, and it may fail due to complexity problems such as floating numbers which are often used in the aeronautics domain. In addition, manually writing temporal specifications is not only time-consuming but also error-prone. To address these challenges, we propose an AGVTS method that can automate the task of generating temporal specifications and verifying aeronautics SCADE models. At first, we define a modular pattern language for precisely expressing Chinese natural language requirements. Then, we present a rule-based translation augmented with BERT, which translates restricted requirements into LTL and CTL. In addition, SCADE model verification is achieved by transforming it into nuXmv which supports both SMT-based and SAT-based verification. Finally, we illustrate a successful application of our methodology with an ejection seat control system, and convince our industrial partners of the usefulness of formal methods for industrial systems.

Keywords: SCADE · Temporal Specification · Pattern-based Language · BERT · nuXmv

1 Introduction

Safety-critical systems (SCS) [20] are the systems whose failure could result in loss of life, substantial economic loss, or damage to the environment. There are many well-known examples in different domains such as avionics, nuclear plants,

Supported by the National Natural Science Foundation of China (62072233, U2241216), Aviation Science Fund of China (201919052002).

© The Author(s) 2025

transportation, and automotive. Formal verification is highly recommended by safety standards, e.g., DO-178C for the avionics domain, in order to ensure the safety of this kind of systems [17]. SCADE is both a formal language [9] and a model-based development environment[1] widely used to build and verify the models of safety-critical system. SCADE provides three modeling styles, i.e., safety state machines, data flow and their combination.

Design Verifier (DV)[2], the formal verification module of SCADE, is a model checker based on a SAT-solver. However, DV cannot adequately express complex temporal specifications [24]. Temporal logics are popular methods for describing complex temporal properties, such as LTL [23], CTL [7] and TCTL [1], etc. There are several related works [12,24] to enhance the verification capability of SCADE, which respectively transform SCADE models into UPPAAL and nuSmv to verify temporal properties. However, these works only consider safety state machine models. Additionally, DV may fail due to complexity problems such as floating numbers which are common in the aeronautics domain.

Moreover, it is always a challenge to manually translate natural language requirements into temporal logic formulae. As natural language is generally informal and ambiguous, this process is error-prone and time-consuming. Existing works on translating natural language requirements into temporal logics can be classified as several categories, such as rule-based [15,29], deep learning [18], and Large Language Models (LLMs) [11]. However, these methods require either manual writing of formal specifications for atomic propositions, or utilizing plenty of patterns, or training with plenty of data. To the best of our knowledge, these works always consider the translation of English, but few consider Chinese, and there is little work focusing on the ejection seat control system domain.

To address the challenges above, we propose an AGVTS method, $\underline{a}$utomatically $\underline{g}$enerating and $\underline{v}$erifying $\underline{t}$emporal $\underline{s}$pecifications for aeronautics SCADE models. The main contributions are summarized as follows:

(1) A modular pattern language (MPL) to precisely express Chinese natural language requirements. Benefiting from the modular structure, users can write requirements in a restricted and composite way with less patterns.
(2) A rule-based method augmented with BERT for automatically translating requirements expressed by MPL into LTL and CTL formulae.
(3) An automated transformation from SCADE to nuXmv that provides SMT-based and SAT-based model checking techniques to verify LTL and CTL properties with floating numbers. Compared with existing works, our transformation covers more modeling styles, such as data flow, safety state machines and their combination.
(4) We apply our method to an industrial ejection seat control system. It successfully translates 124 requirements and verifies the SCADE models of six modules of the system. The result convinces our industrial partners of the usefulness of formal methods to industrial systems.

[1] Ansys SCADE Suite https://www.ansys.com/products/embedded-software/ansys-scade-suite,.
[2] DV is based on Prover Technology proof engines (www.prover.com).

2 Global View of the AGVTS Method

Figure 1 gives an overview of the AGVTS method. AGVTS has three modules shown as follows.

- **Modular Pattern Language (MPL)**: Define a pattern language organized in a modular structure. The syntax and semantics of each pattern guide users to write requirements in a restricted and composite manner.
- **Rule-based Requirements Translator Augmented with BERT**: Parse requirements written in MPL and build pattern structure trees for them. Then generate LTL and CTL formulae through traversing the tree.
- **SCADE2nuXmv Model Transformation**: Transform SCADE models into nuXmv. Subsequently, verify the nuXmv models, and show the verification results and the traceability between requirements and counterexamples.

In the following sections, we will introduce the three modules in detail.

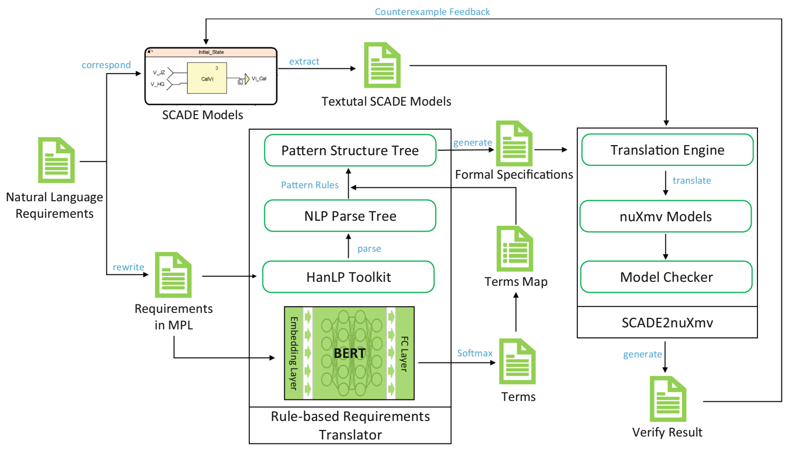


Fig. 1. The Global View of the AGVTS Method

3 Modular Pattern-Based Language

The modular pattern-based language (MPL) focuses on ejection seat control system which usually contains complex computation in the requirements. MPL has three segments: *Atomic Proposition* (*AP*), *Relation*, and *Scope*. Each segment has several patterns. This feature allows users to write requirements in a composite manner with fewer patterns. Additionally, we define the formal syntax and semantics of the patterns[3] As shown in Fig. 2, we first write three atomic

[3] These are shown at https://github.com/yayi-mei/AGVTS.

statements in *AP* patterns. Then the *Relation* patterns are utilized to connect statements for constructing compound statements and *Scope* patterns are added to declare the effective extent of different statements. Finally we obtain the complete requirement. Please note that the *Relation* and *Scope* patterns can be nested in any way.

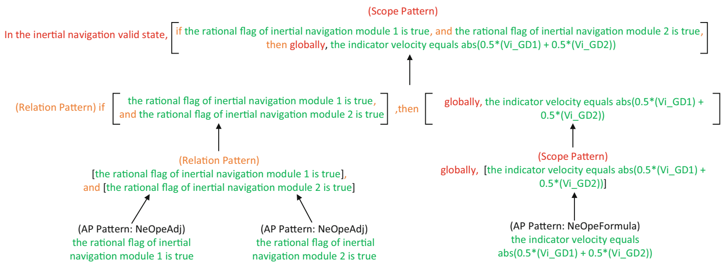

(Scope Pattern)
In the inertial navigation valid state, [if the rational flag of inertial navigation module 1 is true, and the rational flag of inertial navigation module 2 is true, then globally, the indicator velocity equals abs(0.5*(Vi_GD1) + 0.5*(Vi_GD2))]
(Relation Pattern) if [the rational flag of inertial navigation module 1 is true, and the rational flag of inertial navigation module 2 is true] ,then [globally, the indicator velocity equals abs(0.5*(Vi_GD1) + 0.5*(Vi_GD2))]
(Relation Pattern)
[the rational flag of inertial navigation module 1 is true], and [the rational flag of inertial navigation module 2 is true]
(Scope Pattern)
globally, [the indicator velocity equals abs(0.5*(Vi_GD1) + 0.5*(Vi_GD2))]
(AP Pattern: NeOpeAdj)
the rational flag of inertial navigation module 1 is true
(AP Pattern: NeOpeAdj)
the rational flag of inertial navigation module 2 is true
(AP Pattern: NeOpeFormula)
the indicator velocity equals abs(0.5*(Vi_GD1) + 0.5*(Vi_GD2))

Fig. 2. Writing Requirements in a Composite Manner

***AP* Patterns.** *AP* patterns form the basis of MPL, serving to specify an atomic statement on an event or a state of the system. MPL has nine *AP* patterns. Each pattern consists of several *Ingredient* tags we define, such as *Ne* for variables and constants, *Ope* for operators, and *Formula* for complex computation. We have defined seven *Ingredient* tags in total. For Instance, in Fig. 2, "the indicator velocity equals $abs(0.5 * (Vi\_GD1) + 0.5 * (Vi\_GD1))$" is written in "*Ne Ope Formula*" pattern, in which "rational flag of inertial navigation module 1", "equals" and "$abs(0.5 * (Vi\_GD1) + 0.5 * (Vi\_GD1))$" are labeled as *Ne*, *Ope*, and *Formula* respectively.

***Relation* Patterns.** *Relation* patterns describe the temporal or logical relations between atomic or compound statements, which are characterized by different keywords. We have defined six *Relation* patterns: *Simple*, *Response*, *Condition*, *Precedence*, *Conjunction* and *Disjunction*, as shown in Table 1 where ϕ_i $(1 \leq i \leq n)$ denotes a statement. The *Simple* pattern represents the atomic statements in the requirement, which are specified by the *AP* patterns. The *Response* and *Precedence* patterns are introduced from [2]. *Conjunction* and *Disjunction* patterns express the complex nesting relations of a series of statements, as the parentheses are seldom used in Chinese requirements. To support the nesting of *Relation* pattern, we define a priority for each of them.

For example, in Fig. 2, we use *Conjunction* pattern to connect two atomic statements. The compound statement "the rational flag of inertial navigation module 1 is true and the rational flag of inertial navigation module 2 is true" indicates that the rational flags of both inertial navigation modules must be true simultaneously. Moreover, the statement "ϕ_1 weak and ϕ_2, or ϕ_3 weak and ϕ_4" can express "$(\phi_1 \wedge \phi_2) \vee (\phi_3 \wedge \phi_4)$", based on different priorities.

Table 1. Relation Patterns

Pattern Name	Natural Language Format	Meaning	Priority
Simple	ϕ_1	the atomic statement	6
Response	if ϕ_1 holds, then in response ϕ_2 holds	if ϕ_1 holds, then ϕ_2 must holds in the next cycle	1
Condition	if ϕ_1, then ϕ_2	if ϕ_1 holds, then ϕ_2 must holds in the same cycle	0
Precedence	ϕ_2 precedes ϕ_1	if ϕ_1 holds in the future, ϕ_2 must hold at least one time before ϕ_1 holds	1
Conjunction	ϕ_1 and ϕ_2 and $\cdots$ and ϕ_n	all of ϕ_i $(1 \leq i \leq n)$ must hold in the specified cycle	2
	ϕ_1 weak_and ϕ_2 weak_and $\cdots$ weak_and ϕ_n		4
Disjunction	ϕ_1 or ϕ_2 or $\cdots$ or ϕ_n	at least one of ϕ_i $(1 \leq i \leq n)$ holds in the specified cycle	3
	ϕ_1 weak_or ϕ_2 weak_or $\cdots$ weak_or ϕ_n		5

***Scope* Patterns.** *Scope* patterns describe the effective extent of an atomic or compound statement. We have defined 11 *Scope* patterns, such as "Globally", "In x state", and "Every n cycles", where x denotes a state name and n denotes a positive integer. As shown in Fig. 2, the scope "In the inertial navigation valid state" expresses that the property stated subsequently must hold when the system is in the "inertial navigation valid" state.

4 Rule-Based Translation Augmented with BERT

In this section, we will introduce the rule-based translation augmented with BERT method which translates requirements into LTL and CTL specifications. As shown in Fig. 3, to illustrate the workflow of the method, we consider the following requirement:

Example 1. Globally, if the state of the system is calculating angle by two inertial navigation modules, then globally the input angle always equals 0.5 times the sum of the angles in inertial modules 1 and 2.

In the following paragraphs, we will represent the details of each step.

Rewriting & Pre-processing. Natural language are usually ambiguous, so we rewrite the requirement in MPL first. Additionally, the complex calculations expressed in natural language should be replaced with corresponding formal expression to simplify the translation. For example, the statement "0.5 times the sum of the angles in inertial modules 1 and 2" in *Example 1* is replaced by "$(0.5 * (angle\_GD1 + angle\_GD2))$". The *Example 1* is rewritten as "Globally,

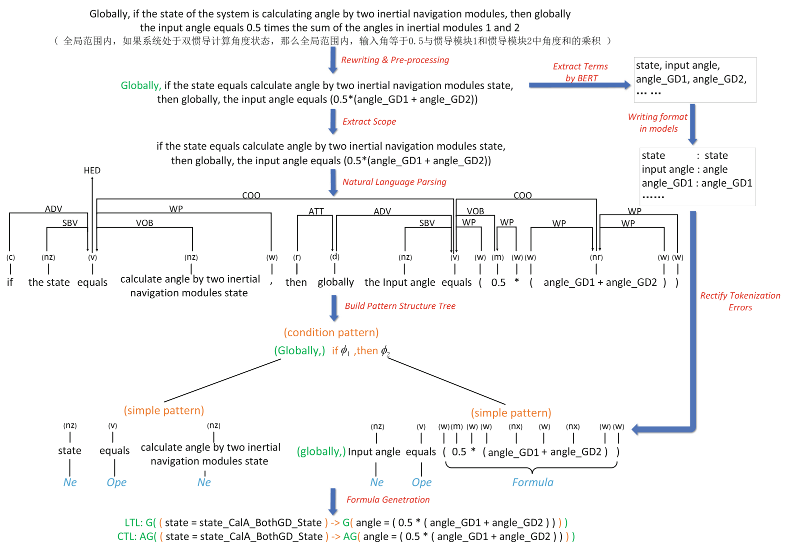

Globally, if the state of the system is calculating angle by two inertial navigation modules, then globally the input angle equals 0.5 times the sum of the angles in inertial modules 1 and 2
（全局范围内，如果系统处于双惯导计算角度状态，那么全局范围内，输入角等于0.5与惯导模块1和惯导模块2中角度和的乘积）
Rewriting & Pre-processing
Globally, if the state equals calculate angle by two inertial navigation modules state, then globally, the input angle equals (0.5*(angle_GD1 + angle_GD2))
Extract Terms by BERT
state, input angle, angle_GD1, angle_GD2, … …
Writing format in models
state : state
input angle : angle
angle_GD1 : angle_GD1
……
Extract Scope
if the state equals calculate angle by two inertial navigation modules state, then globally, the input angle equals (0.5*(angle_GD1 + angle_GD2))
Natural Language Parsing
HED
COO
ADV
SBV
WP
VOB
ATT
VOB
WP
if
the state
equals
calculate angle by two inertial navigation modules state
,
then
globally
the Input angle
equals
(
0.5
*
(
angle_GD1 + angle_GD2
)
)
Rectify Tokenization Errors
Build Pattern Structure Tree
(condition pattern)
(Globally,) if ϕ_1 ,then ϕ_2
(simple pattern)
state
equals
calculate angle by two inertial navigation modules state
Ne
Ope
Ne
(simple pattern)
(globally,) Input angle equals (0.5 * (angle_GD1 + angle_GD2))
Ne
Ope
Formula
Formula Genetration
LTL: G((state = state_CalA_BothGD_State) -> G(angle = (0.5 * (angle_GD1 + angle_GD2))))
CTL: AG((state = state_CalA_BothGD_State) -> AG(angle = (0.5 * (angle_GD1 + angle_GD2))))

Fig. 3. Overview of the Rule-Based Translation Augmented with BERT

if the state equals calculate angle by two inertial navigation modules state, then globally, the input angle equals $(0.5 * (angle\_GD1 + angle\_GD2))$".

Extract *Scopes* & Natural Language Parsing. The accuracy of a NLP parser decreases as the length of the requirement increases, so the requirement should be as concise as possible. Since the expressions of *Scope* patterns in MPL are fixed, we extract them with regular expression before parsing the rewritten requirement. For example, we extract "*Globally,*" from *Example 1*. Then HanLP[4], a Chinese NLP toolkit, is leveraged to parse the requirement, including tokenization, POS tagging, and dependency relations analysis.

Extracting Terms by BERT. Compared with English, Chinese natural language lacks separators to distinguish words. Therefore, tokenization is the initial task when parsing Chinese with NLP techniques. However, incorrect tokenization may occur. Clearly, an incorrect tokenization will finally lead to a wrong parsing result. For example, "$angle\_GD1 + angle\_GD2$" in *Example 1* is recognized as one token.

To solve this problem, we implement a *Term Extractor* to extract terms from requirements, which assists in improving the accuracy of word segmentation and correcting its results. Term extraction essentially classifies each token in the requirement into three categories: the beginning of the term, the body of the term, and non-term. Therefore, we build a deep learning model, as shown in Fig. 4, to complete this task. This model first utilizes BERT [13], a pre-trained language model, to extract text features from the requirements. Then a fully

[4] https://github.com/hankcs/HanLP/.

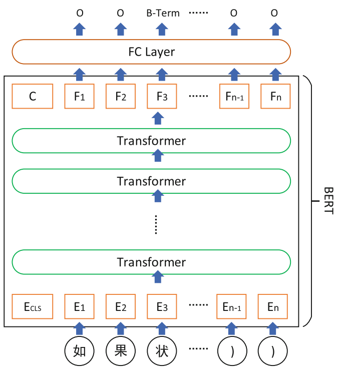
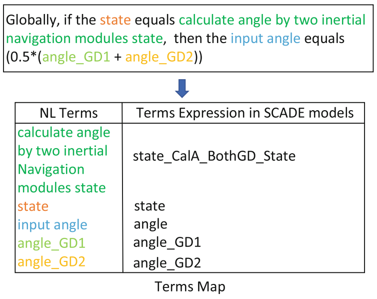


Fig. 4. The Structure of the BERT-Based Model



Globally, if the state equals calculate angle by two inertial navigation modules state, then the input angle equals (0.5*(angle_GD1 + angle_GD2))

NL Terms	Terms Expression in SCADE models
calculate angle by two inertial Navigation modules state	state_CalA_BothGD_State
state	state
input angle	angle
angle_GD1	angle_GD1
angle_GD2	angle_GD2

Terms Map

Fig. 5. Examples of a Terms Map

connected layer (FC Layer) is utilized to calculate the probability of each token belonging to different categories based on the text features. Finally, each token is labeled as the category with the highest probability.

The extracted terms serve two purposes. One is to expand the tokenization lexicon of HanLP, which reduces the probability of incorrect tokenization. The other is to rectify the potential errors in the parsing result of HanLP. In addition, we construct a *terms map* in order to match the extracted terms with their corresponding variables or constants in the SCADE model, as shown in Fig. 5.

Build Pattern Structure Tree & Rectify NLP Errors. After getting the NLP parsing result, we recursively construct a pattern structure tree for each statement in the requirement. The whole pattern structure tree describes the nesting relations among the *Relation* patterns present in the requirement. Each node of the tree corresponds to a statement in the requirement, including the *Relation* pattern of the statement, the keywords of the *Relation* pattern, and the *Scope* pattern of the statement. Please note that each leaf node of this tree corresponds to an atomic statement in the requirement.

Each *Relation* pattern recorded by the tree node represents the one with the highest priority in its corresponding statement. To determine the *Relation* pattern with the highest priority, we utilize tokenization and lexical matching to find the keywords of *Relation* patterns. Then we compare the priorities of the patterns they belong to, as shown in Table 2.

In cases where no keywords are identified, the pattern of the statement is the *Simple* pattern. Then we identify the *AP* pattern of the statement by determining the *Ingredient* tag of each token and combining the tags in order. The task is performed by tokenization, lexical matching and POS tagging. Additionally, we utilize domain lexicons which include verbs, adjectives, operators and functions in the requirement to filter useless tokens. The extracted terms are leveraged to enhance the NLP result in this process. That is, for tokens categorized as "noun", we substitute the original token with the term that matches the longest consecutive characters in the requirement.

As shown in Fig. 3, the *Condition* pattern recorded by the root node of the pattern structure tree has the highest priority in *Example 1*. Both of its children are atomic statements, so the corresponding nodes of these statements record their *AP* and *Scope* patterns. In this process, the incorrect token "$angle\_GD1 + angle\_GD2$" is rectified to "$angle\_GD1$", "$angle\_DD2$" and the operator "+".

Formula Generation. The last step is generating formal specifications by composing formal expressions of each node on the pattern structure tree in post order. The mapping of our *Relation* patterns to LTL and CTL are shown in Table 2, and the mapping rules of *AP* patterns and *Scope* patterns are illustrated in the appendix. These rules strictly follow the formal semantics of MPL.

Table 2. Mapping Rules of *Relation* Patterns (Except for *Simple* Pattern)

Pattern Name	LTL Formula	CTL Formula
Response	$\phi_1 \rightarrow X\,\phi_2$	$\phi_1 \rightarrow AX\,\phi_2$
Condition	$\phi_1 \rightarrow \phi_2$	$\phi_1 \rightarrow \phi_2$
Precedence	$F\,\phi_1 \rightarrow (!\,\phi_1\,U\,(\phi_2 \wedge !\,\phi_1))$	$AF\,\phi_1 \rightarrow A(!\,\phi_1\,U\,(\phi_2 \wedge !\,\phi_1))$
Conjunction	$\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$	$\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$
Disjunction	$\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n$	$\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n$

Based on the mapping rules above, as shown in Fig. 3, we first translate the leaf nodes on the pattern structure tree into AP formulae. For instance, "state equals calculate angle by two inertial navigation modules state" is translated to "$state = state\_CalA\_BothGD\_State$". Secondly, the generated formulae are connected by "$\rightarrow$" which is translated from the keywords recorded by the root node. During the translation, the *Scope* patterns ,"globally", are translated into corresponding temporal operators and added to the corresponding formulae. The LTL and CTL formulae are shown as follows:

$$LTL : G((state = state\_CalA\_BothGD\_State) \rightarrow G(angle = (0.5 * (angle\_GD1 + angle\_GD2)))) \tag{1}$$

$$CTL : AG((state = state\_CalA\_BothGD\_State) \rightarrow AG(angle = (0.5 * (angle\_GD1 + angle\_GD2)))) \tag{2}$$

5 SCADE2nuXmv Model Transformation

In this section, we will introduce the automated transformation from SCADE to nuXmv. Figure 6 shows the overview of SCADE2nuXmv. We first extract the textual representation of the SCADE model through SCADE IDE. Subsequently, use ANTLR, a toolkit for lexical analysis and syntax analysis, to build a syntax tree for it. Then generate target nuXmv models based on the syntax tree. Finally, we employ the model checker of nuXmv to verify the generated model. The traceability between the execution trace of counterexamples, and corresponding formulae and requirements is also generated.

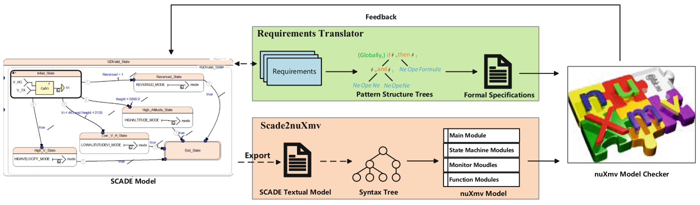
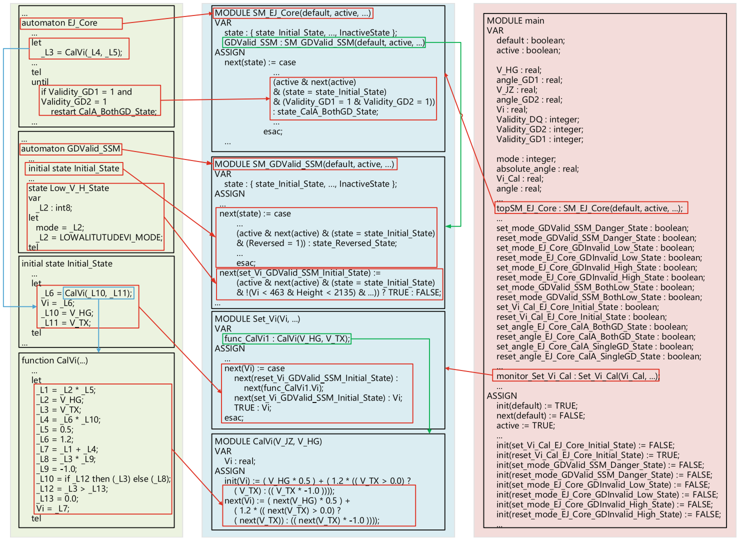


Fig. 6. SCADE Models Verification Achieved by SCADE2nuXmv

The reasons why we choose nuXmv are the ability to express hierarchical models, support for both SMT-based and SAT-based verification, and the verification of both infinite-state and finite-state systems. For instance, the middle and right columns in Fig. 7 construct a nuXmv model that contains hierarchical state machine. The *SM_EJ_Core* is the top state machine of the model. It declares the sub-state machine *SM_GDValid_SSM* module, which selects strategy according to the different variables (e.g., *Vi*), in its *VAR* part.



```
...
automaton EJ_Core
   ...
   let
      _L3 = CalVi(_L4, _L5);
   ...
   tel
   until
      if Validity_GD1 = 1 and
      Validity_GD2 = 1
         restart CalA_BothGD_State;
   ...
```

```
...
automaton GDValid_SSM
   ...
   initial state Initial_State
   ...
   state Low_V_H_State
   var
      _L2 : int8;
   let
      mode = _L2;
      _L2 = LOWALITUTUDEVI_MODE;
   tel
```

```
initial state Initial_State
   ...
   let
      _L6 = CalVi(_L10, _L11);
      Vi = _L6;
      _L10 = V_HG;
      _L11 = V_TX;
   tel
...
```

```
function CalVi(...)
   ...
   let
    _L1 = _L2 * _L5;
    _L2 = V_HG;
    _L3 = V_TX;
    _L4 = _L6 * _L10;
    _L5 = 0.5;
    _L6 = 1.2;
    _L7 = _L1 + _L4;
    _L8 = _L3 * _L9;
    _L9 = -1.0;
    _L10 = if _L12 then (_L3) else (_L8);
    _L12 = _L3 > _L13;
    _L13 = 0.0;
    Vi = _L7;
   tel
```

```
MODULE SM_EJ_Core(default, active, ...)
VAR
   state : { state_Initial_State, ..., InactiveState };
   GDValid_SSM : SM_GDValid_SSM(default, active, ...)
ASSIGN
   next(state) := case
                    ...
                    (active & next(active)
                    & (state = state_Initial_State)
                    & (Validity_GD1 = 1 & Validity_GD2 = 1))
                    : state_CalA_BothGD_State;
                    ...
                 esac;
   ...
```

```
MODULE SM_GDValid_SSM(default, active, ...)
VAR
   state : { state_Initial_State, ..., InactiveState };
ASSIGN
 ...
 next(state) := case
       ...
       (active & next(active) & (state = state_Initial_State)
       & (Reversed = 1)) : state_Reversed_State;
       ...
       esac;
 next(set_Vi_GDValid_SSM_Initial_State) :=
       (active & next(active) & (state = state_Initial_State)
       & !(Vi < 463 & Height < 2135) & ...)) ? TRUE : FALSE;
...
```

```
MODULE Set_Vi(Vi, ...)
VAR
   func_CalVi1 : CalVi(V_HG, V_TX);
ASSIGN
   ...
   next(Vi) := case
      next(reset_Vi_GDValid_SSM_Initial_State) :
         next(func_CalVi1.Vi);
      next(set_Vi_GDValid_SSM_Initial_State) : Vi;
      TRUE : Vi;
   esac;
```

```
MODULE CalVi(V_JZ, V_HG)
VAR
   Vi : real;
ASSIGN
   init(Vi) := ( V_HG * 0.5 ) + ( 1.2 * (( V_TX > 0.0) ?
      ( V_TX) : (( V_TX * -1.0 ))));
   next(Vi) := ( next(V_HG) * 0.5 ) +
      ( 1.2 * (( next(V_TX) > 0.0) ?
      ( next(V_TX)) : (( next(V_TX) * -1.0 ))));
```

```
MODULE main
VAR
   default : boolean;
   active : boolean;

   V_HG : real;
   angle_GD1 : real;
   V_JZ : real;
   angle_GD2 : real;
   Vi : real;
   Validity_DQ : integer;
   Validity_GD2 : integer;
   Validity_GD1 : integer;

   mode : integer;
   absolute_angle : real;
   Vi_Cal : real;
   angle : real;
   ...
   topSM_EJ_Core : SM_EJ_Core(default, active, ...);
   ...
   set_mode_GDValid_SSM_Danger_State : boolean;
   reset_mode_GDValid_SSM_Danger_State : boolean;
   set_mode_EJ_Core_GDInvalid_Low_State : boolean;
   reset_mode_EJ_Core_GDInvalid_Low_State : boolean;
   set_mode_EJ_Core_GDInvalid_High_State : boolean;
   reset_mode_EJ_Core_GDInvalid_High_State : boolean;
   set_mode_GDValid_SSM_BothLow_State : boolean;
   reset_mode_GDValid_SSM_BothLow_State : boolean;
   set_Vi_Cal_EJ_Core_Initial_State : boolean;
   reset_Vi_Cal_EJ_Core_Initial_State : boolean;
   set_angle_EJ_Core_CalA_BothGD_State : boolean;
   reset_angle_EJ_Core_CalA_BothGD_State : boolean;
   set_angle_EJ_Core_CalA_SingleGD_State : boolean;
   reset_angle_EJ_Core_CalA_SingleGD_State : boolean;
   ...
   monitor_Set_Vi_Cal : Set_Vi_Cal(Vi_Cal, ...);
   ...
ASSIGN
   init(default) := TRUE;
   next(default) := FALSE;
   active := TRUE;
   ...
   init(set_Vi_Cal_EJ_Core_Initial_State) := FALSE;
   init(reset_Vi_Cal_EJ_Core_Initial_State) := TRUE;
   init(set_mode_GDValid_SSM_Danger_State) := FALSE;
   init(reset_mode_GDValid_SSM_Danger_State) := FALSE;
   init(set_mode_EJ_Core_GDInvalid_Low_State) := FALSE;
   init(reset_mode_EJ_Core_GDInvalid_Low_State) := FALSE;
   init(set_mode_EJ_Core_GDInvalid_High_State) := FALSE;
   init(reset_mode_EJ_Core_GDInvalid_High_State) := FALSE;
   ...
```

Fig. 7. The Translation from SCADE to nuXmv

The nuXmv models generated by our method contains four module types: *Monitor*, *State Machine*, *Function* and *Main*. The *Monitor* module implements the monitor mechanism. The *State Machine* module and *Function* module respectively represent safety state machine and data flow in SCADE models. The *Main* module is the entry of target models. In the following, we will introduce these modules in detail.

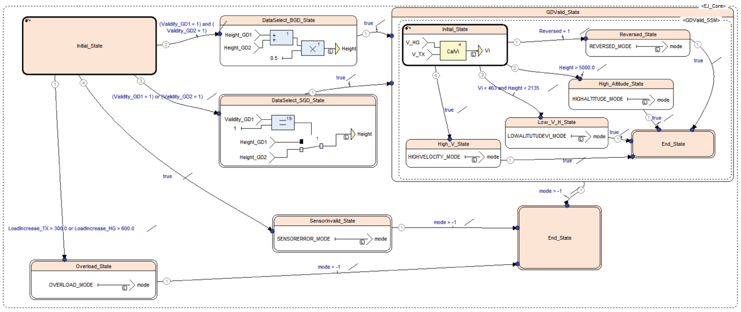


Fig. 8. The SCADE Model Example

Monitor Module. The language supported by nuXmv forbids multiple assignments to a variable [4], which is common in safety state machine, so we created *Monitor* modules inspired by [8] to manipulate the assignment of output and local variables. Each *Monitor* module incorporates two "case" expressions with multiple execution branches, to assign the value of the monitored variable. One initializes the monitored variable, while the other updates it after the first cycle.

Each "case" expression branch corresponds to an assignment expression of the monitored variable in the original SCADE model. To determine the branch to execute, we create *monitor variables* similar to the ones in [8]. Each *monitored variable* triggers a branch when it becomes true. To distinguish between assignment expressions related to state transitions, we employ two types of monitor variables, $reset\_x\_s$ and set_x_s, where x denotes the monitored variable and s denotes the state assigning the value of x. When the state machine transitions to s, $reset\_x\_s$ turns TRUE. If none of the transition conditions of state s are met, set_x_s becomes TRUE. We further replace variables prefixed with "_L", which represent connection lines in the SCADE model, with their equivalent variables to reduce the state space. For example, the "Set_Vi" in Fig. 7 is an example of such a module. The "case" expression assigns the value of "Vi" according to the monitor variables of each branch.

State Machine Module. Each *State Machine* module corresponds to one state machine in the original SCADE model. It records the state transition, assigns monitor variables and instantiates submachines. Such modules also implement the hierarchical structure in safety state machine utilizing *active* and *default* variables, which respectively denote the activation of the state machine and whether the state machine transitions from inactive to active in the cycle. For example, the "*SM_GDValid_SSM*" module in Fig. 7 is a *State Machine* module, representing the state machine "*GDValid_SSM*" in Fig. 8. The first "case" expression depicts the state transition, while subsequent one assigns monitor variables related to it.

Function Module. A *Function* module corresponds to a data flow operator in the original SCADE model. Each output variable of it is defined by two expressions in the corresponding *Function* module. One, the same as the expression in the SCADE model, assigns its initial value. The other, similar to the expression in the SCADE model but with each variable enclosed by a *next* operator, assigns the value of the variable after the first cycle. After the translation, we further replace the temporary variables "_Li" (where i is a positive integer), that represent connection lines in the SCADE model, with their equivalent variables to reduce the state space. For example, the "*CalVi*" module in Fig. 7 is a *Function* module. The temporary variable "_Li" is replaced with its equivalent variable, e.g., "_L2" is replaced with "V_HG".

Main Module. Each nuXmv model has one *Main* module which instantiates the interface of the SCADE model, the local variables, all monitor variables and modules, and the top state machine. It also assigns the initial value of all monitoring variables. Additionally, a variable equalling the state of the top state machine is defined in the *Main* module, allowing users to verify specifications related to a specific state. This variable only exists when the original SCADE model contains safety state machines.

As mentioned in Sect. 1, SCADE provides three modeling styles, i.e., safety state machines, data flow and their combination. When the SCADE model is data flow style, its nuXmv model contains *Main Module* and *Function Module*. On the contrary, when the SCADE model is safety state machine style, we use *State Machine* and *Main* modules to construct the target nuXmv models. In addition, we use all the above four types of modules to construct the target nuXmv model for the SCADE model that combines data flow and safety state machine.

6 Industrial Case Studies and Evaluation

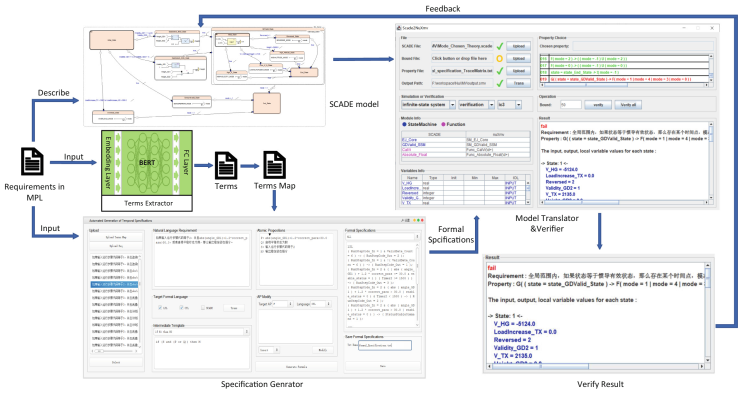
6.1 Ejection Seat Control System

Ejection seats must eject pilots out of the cockpit when the pilots pull the switch and open the parachute at an appropriate time to safely send pilots back to the ground. When the seat is ejected from the cockpit, the control system in the seat chooses different control strategies as the environment varies. During this process, it avoids rotation, excessive loads and wrong movement direction. When the seat is in a non-upright position, the control system adjusts the attitude of the seat.

As a typical safety-critical software, SCADE is suitable to model this control system. Moreover, the control system controls all subsystems of the ejection seat to faithfully implement the chosen strategy. Each step in the strategy has a strict execution order. This feature makes temporal logic suitable for describing the specifications of its requirements.

6.2 Implementation and Experiments

We have developed a tool chain to implement the AGVTS method in this paper. To verify the effectiveness of AGVTS, we utilize the tool chain to verify six modules in the ejection seat control system against a set of 124 requirements.



```
Result
fail
Requirement : 全局范围内，如果状态等于惯导有效状态，那么存在某个时间点，模
Property : G( ( state = state_GDValid_State ) -> F( mode = 1 | mode = 4 | mode =

The input, output, local variable values for each state :

-> State: 1 <-
  V_HG = -5124.0
  LoadIncrease_TX = 0.0
  Reversed = 2
  Validity_GD2 = 1
  V_TX = 2135.0
```

Fig. 9. Implementation of the Approach

As shown in Fig. 9, the tool chain first uses an *BERT-based terms extractor*, which is fine-tuned on the requirements of the ejection seat control system, to extract terms from the requirements written in MPL. Industrial engineers subsequently confirm and rectify the errors in the extracted terms. Then they build a *terms map* based on the terms. Thirdly, the *specification generator* reads the *terms map* and converts the requirements written in MPL into LTL and CTL formulae. Finally a *model transformer & verifier* transforms the SCADE model into nuXmv and verifies the generated formulae against the nuXmv model. Additionally, it generates the traceability between the execution trace of counterexamples, and corresponding formulae and requirements to help rectify the original model.

To ensure the effectiveness of the tool chain, we invite several formal experts to confirm whether the generated LTL and CTL formulas accurately capture the intent of the requirements. Then the experts check the generated nuXmv models to determine whether these models faithfully implement the functions of the original models.

6.3 Evaluation

The major objective of this subsection is to evaluate the effectiveness of our method. We will explain the evaluation from three perspectives: Terms Extraction, Requirements Translation and Model Verification.

Table 3. The Statistics of Our BERT-Based Model

Data Set	Precision/%	Recall/%	F_1/%
Fine-tuned Test Set	93.94	95.88	94.90
Requirements of six modules	86.81	90.80	88.76

Terms Extraction. As shown in Table 3, the BERT-based terms extractor achieves a precision of 93.94% and a recall of 95.88% on the fine-tuned test set. The 124 requirements used in our experiment contain 87 terms. The BERT-based terms extractor extracts 91 terms, 79 of which are correct. Error-extracted terms can be divided into two categories. One is common nouns (e.g., "cycle") that do not belong to ejection seat control system. The other is the segment of larger terms, for example "Inertial Navigation Module" is a wrong term split from "Valid Inertial Navigation Module State". However, these can be easily identified and rectified by engineers.

Table 4. The Statistics of the Verification Results

Modules	Reqs	AGVTS (Verify)	Wrong Req
Mode Selection	21	21	0
Ejection Judge	18	18	2
High Altitude Mode	26	26	0
Velocity Control Mode	20	16	0
Overload Mode	24	24	0
Stability Control	15	15	0

Requirements Translation. Our method successfully translates all the pre-processed requirements written in the pattern based language into LTL and CTL formulae. Table 4 shows the translation statistics data. In the following we provide two requirements described in MPL to show the translation.

Example 2. "Globally, in the Inertial Navigation Valid State, the mode will be set to *reverse* mode, or the mode will be set to *low velocity low altitude* mode, or the mode will be set to *high velocity* mode, or the mode will be set to *high altitude* mode".

This is a property used to limit the output of the system whose corresponding LTL and CTL formulae are shown as follows.

$$LTL : G((state = state\_GDValid\_State) \rightarrow F(mode = 4 \,|\, mode = 6 \,|\, mode = 0 \,|\, mode = 5)) \tag{3}$$

$$CTL : AG((state = state\_GDValid\_State) \rightarrow AF(mode = 4 \,|\, mode = 6 \,|\, mode = 0 \,|\, mode = 5)) \tag{4}$$

Please note that, the terms map has matched the *reverse* mode, *low velocity low altitude* mode, *high velocity* mode, and *high altitude* mode to the constants defined in the SCADE model. For instance "the mode will be set to *reverse* mode" is translated to "$mode = 4$".

Example 3. "If the rational flag of inertial navigation module 1 is true, and the rational flag of inertial navigation module 2 is true, then in the next cycle, state will be set to calculate_angle_Both_Inertial_Valid state".

As the statement does not have a *Scope*, it is only checked in the first cycle. Formula (5) and (6) represent the LTL and CTL formula of this requirement.

$$LTL : (Validity\_GD1 = 1 \,\&\, Validity\_GD2 = 1) \rightarrow X(state = state\_CalA\_BothGD\_State) \tag{5}$$

$$CTL : (Validity\_GD1 = 1 \,\&\, Validity\_GD2 = 1) \rightarrow AX(state = state\_CalA\_BothGD\_State) \tag{6}$$

Model Verification. The SCADE models of the six modules contain three types of structure: data flow, safety state machine and their combination. Our method successfully transforms them. Then we verify the nuXmv models with the generated formulae.

During the verification, two bugs caused by the "case" statement are found. The first one is that the front case condition covers the latter case condition. The second one is caused by two identical case condition, but their actions are different. This result shows that our method for verification is effective. Additionally, our method verifies the SCADE models of the six modules in a shorter time than DV.

Four requirements of the *Velocity Control* mode contain nonlinear calculations, such as square root, that cannot be verified by SCADE DV and our method. Note that, when the SCADE models contain unbounded integers and real numbers, nuXmv just verify the LTL specifications.

7 Related Work

Formal Specification Generation. As writing formal specifications is time-consuming and error-prone, [14,19] propose a set of patterns corresponding to different scenarios and their formal semantics to guide users to write formal specifications. [15] develops a SpeAR tool which translates requirements written in pattern language into PLTL. [10,16,21] provide a framework to translate requirements written in the pattern language FRETISH into formal specifications. [2] proposes a framework combining existing classical patterns. [22,26,29] utilize NLP techniques, such as POS tagging and dependency parsing, to pre-process requirements. Then they define translation rules to generate formal specifications based on the pre-processed requirements. [18,25] treat the translation from natural language to formal specifications as a machine translation task and utilize deep learning models to solve it. [5,11] utilize Large Language Models (LLMs) to complete the translation. [11] decomposes the natural language input into sub-translations by utilizing LLMs. However, these methods require either manual writing of formal specifications for atomic propositions, or utilizing plenty of patterns, or training with plenty of data.

Verification of SCADE Models. As SCADE DV cannot adequately express complex temporal specifications and it may fail due to complexity problems such as floating numbers, there are several related works to enhance the verification capability of SCADE. [12] transforms SCADE models into UPPAAL and verify the liveness properties in TCTL. However, it may fail when the original SCADE model contains hierarchical structure. [24] transforms SCADE models into nuSmv [6] models for verification, but limited to safety state machines and incapable of verifying infinte-state system. [3] introduces LAMA as an intermediate language into which SCADE programs can be transformed and which easily can be transformed into SMT solver instances. However, the method performs worse than DV.

8 Conclusion and Future Work

We propose an AGVTS method for automatically generating temporal specifications and verifying aeronautics SCADE models. AGVTS begins by defining a modular pattern language (MPL) to express Chinese requirements precisely. Then it uses a rule-based method augmented with BERT to translate requirements in MPL into LTL and CTL formulae. Finally it verifies SCADE models by transforming them into nuXmv which supports SMT-based and SAT-based verification. The method is applied to an ejection seat control system.

We are currently incorporating LLMs to enhance the capability of terms extraction and the process of parsing requirements. The patterns will be refined to cover more domains and enable users to articulate requirements with greater flexibility. Additionally, we will consider theorem prover Coq to prove the correctness of formula generation and the transformation from SCADE to nuXmv, based on our previous researches [27,28]. Improving the verification capability of nuXmv is also an interesting work.

The details of patterns, translation algorithms of SCADE2nuXmv and appendix are provided at https://github.com/yayi-mei/AGVTS.

References

1. Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Inf. Comput. **104**(1), 2–34 (1993)
2. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar. IEEE Trans. Softw. Eng. **41**(7), 620–638 (2015)

3. Basold, H., Günther, H., Huhn, M., Milius, S.: An open alternative for SMT-based verification of scade models. In: Formal Methods for Industrial Critical Systems: 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings 19, pp. 124–139. Springer (2014). https://doi.org/10.1007/978-3-319-10702-8_9
4. Bozzano, M., et al.: nuxmv 2.0. 0 user manual. fondazione bruno kessler. Tech. rep., Technical report, Trento, Italy (2019)
5. Chen, Y., Gandhi, R., Zhang, Y., Fan, C.: NL2TL: Transforming natural languages to temporal logics using large language models. In: Bouamor, H., Pino, J., Bali, K. (eds.) Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp. 15880–15903. Association for Computational Linguistics, Singapore (2023)
6. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model verifier. In: Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11, pp. 495–499. Springer (1999). https://doi.org/10.1007/s100090050046
7. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. (TOPLAS) **8**(2), 244–263 (1986)
8. Clarke, E.M., Heinle, W.: Modular Translation of Statecharts to SMV. Tech. rep, Citeseer (2000)
9. Colaço, J.L., Pagano, B., Pouzet, M.: Scade 6: A formal language for embedded critical software development. In: 2017 International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 1–11. IEEE (2017)
10. Conrad, E., Titolo, L., Giannakopoulou, D., Pressburger, T., Dutle, A.: A compositional proof framework for fretish requirements. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 68–81 (2022)
11. Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: nl2spec: interactively translating unstructured natural language to temporal logics with large language models. In: International Conference on Computer Aided Verification, pp. 383–396. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_18
12. Daskaya, I., Huhn, M., Milius, S.: Formal safety analysis in industrial practice. In: Formal Methods for Industrial Critical Systems: 16th International Workshop, FMICS 2011, Trento, Italy, August 29-30, 2011. Proceedings 16, pp. 68–84. Springer (2011). https://doi.org/10.1007/978-3-642-24431-5_7
13. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
14. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st international conference on Software engineering, pp. 411–420 (1999)
15. Fifarek, A.W., Wagner, L.G., Hoffman, J.A., Rodes, B.D., Aiello, M.A., Davis, J.A.: Spear v2. 0: formalized past LTL specification and analysis of requirements. In: NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings 9, pp. 420–426. Springer (2017). https://doi.org/10.1007/978-3-319-57288-8_30
16. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J.: Automated formalization of structured natural language requirements. Inf. Softw. Technol. **137**, 106590 (2021). https://doi.org/10.1016/j.infsof.2021.106590

17. Gleirscher, M., van de Pol, J., Woodcock, J.: A manifesto for applicable formal methods. Softw. Syst. Model. **22**(6), 1737–1749 (2023)
18. He, J., Bartocci, E., Ničković, D., Isakovic, H., Grosu, R.: Deepstl: from English requirements to signal temporal logic. In: Proceedings of the 44th International Conference on Software Engineering, pp. 610–622 (2022)
19. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software Engineering, pp. 372–381 (2005)
20. Leveson, N.G.: Engineering a safer world: systems thinking applied to safety, The MIT Press (2016)
21. Mavridou, A., Bourbouh, H., Garoche, P.L., Giannakopoulou, D., Pessburger, T., Schumann, J.: Bridging the gap between requirements and Simulink model analysis. In: Joint 26th International Conference on Requirements Engineering: Foundation for Software Quality Workshops, Doctoral Symposium, Live Studies Track, and Poster Track (2020)
22. Nayak, A., Timmapathini, H.P., Murali, V., Ponnalagu, K., Venkoparao, V.G., Post, A.: Req2spec: transforming software requirements into formal specifications using natural language processing. In: International Working Conference on Requirements Engineering: Foundation for Software Quality, pp. 87–95. Springer (2022). https://doi.org/10.1007/978-3-030-98464-9_8
23. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), pp. 46–57. IEEE (1977)
24. Shi, J., Shi, J., Huang, Y., Xiong, J., She, Q.: Scade2nu: A tool for verifying safety requirements of scade models with temporal specifications. In: REFSQ Workshops (2019)
25. Wang, C., Ross, C., Kuo, Y.L., Katz, B., Barbu, A.: Learning a natural-language to LTL executable semantic parser for grounded robotics. In: Conference on Robot Learning, pp. 1706–1718. PMLR (2021)
26. Yan, R., Cheng, C.H., Chai, Y.: Formal consistency checking over specifications in natural languages. In: 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1677–1682. IEEE (2015)
27. Yang, Z., Bodeveix, J., Filali, M.: Towards a simple and safe objective caml compiling framework for the synchronous language SIGNAL. Frontiers Comput. Sci. **13**(4), 715–734 (2019)
28. Yang, Z., Bodeveix, J., Filali, M., Hu, K., Zhao, Y., Ma, D.: Towards a verified compiler prototype for the synchronous language SIGNAL. Frontiers Comput. Sci. **10**(1), 37–53 (2016)
29. Zhang, S., Zhai, J., Bu, L., Chen, M., Wang, L., Li, X.: Automated generation of LTL specifications for smart home IoT using natural language. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 622–625. IEEE (2020)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Code-Level Safety Verification for Automated Driving: A Case Study

Vladislav Nenchev[1](✉), Calum Imrie[2], Simos Gerasimou[2], and Radu Calinescu[2]

[1] BMW Group, Petuelring 130, 80809 Munich, Germany
vladislav.nenchev@bmw.de
[2] Department of Computer Science, University of York, York, UK
{calum.imrie,simos.gerasimou,radu.calinescu}@york.ac.uk

Abstract. The formal safety analysis of automated driving vehicles poses unique challenges due to their dynamic operating conditions and significant complexity. This paper presents a case study of applying formal safety verification to adaptive cruise controllers. Unlike the majority of existing verification approaches in the automotive domain, which only analyze (potentially imperfect) controller models, employ simulation to find counter-examples or use online monitors for runtime verification, our method verifies controllers at code level by utilizing bounded model checking. Verification is performed against an invariant set derived from formal specifications and an analytical model of the required behavior. For neural network controllers, we propose a scalable three-step decomposition, which additionally uses a neural network verifier. We show that both traditionally implemented as well as neural network controllers are verified within minutes. The dual focus on formal safety and implementation verification provides a comprehensive framework applicable to similar cyber-physical systems.

Keywords: Model-based verification · Formal safety verification · Deep neural network verification · Adaptive cruise control · Automated driving

1 Introduction

Ensuring the safe operation of self-driving cars requires controlling the vehicle by software, crafted by numerous developers utilizing complex architectures, various programming languages, middleware etc. Automating the validation and verification of this software is crucial for certification and a rapid release cycle. However, proving safety for all possible driving scenarios in the allowed operation domain has proven to be very challenging. A safe Adaptive Cruise Controller (ACC) has to keep a suitable distance to all relevant target objects, such that the automated driving vehicle maintains a safe distance even in the presence of uncertainties. Interestingly, some common ACC solutions have been shown to be

© The Author(s) 2025

unstable with respect to formal models [43]. To guarantee the safe operation of a controller, in practice, one often resorts to a tailored redundant architecture, as well as exhaustive simulation and testing. While the latter methods can be performed automatically even when the controllers under test utilize Deep Neural Networks (DNNs) [46], they are neither sound (i.e., some bugs may be missed) nor complete (i.e., not every bug report corresponds to a real bug).

To address this limitation of current ACC verification approaches, we propose a formal framework for the development-time safety verification of adaptive cruise controllers using set-invariance methods. We model the motion of the vehicle and the relevant target object to keep a safe distance from as a discrete-time linear system subject to bounded control inputs. We specify the safety requirements as infinite-time collision avoidance, while restricting the cruise speed of the vehicles to suitable ranges. This allows us to define an operation set corresponding to the safety specification. Then, we compute a safe set within the operation set based on a Controlled Invariant Set (CIS) for the discrete-time linear system. The CIS is used to verify (or falsify) the closed-loop operation of several ACC implementations offline. For traditional controllers, a Bounded Model Checker (BMC) is utilized to prove safety. For DNN-based controllers, we propose a new hybrid verification approach based on decomposition: we verify the deployment code (loading the DNN, DNN-based inference, etc.) using a BMC, and the actual DNN with a dedicated neural network verification tool (e.g., Marabou [25]). Our case study shows that the proposed framework can verify (or falsify) the safety of both traditional and DNN-based ACC implementations within minutes on a standard workstation. The considered controllers are commonly employed in contemporary industrial and research approaches [13] for automated driving, including a model predictive controller [31] with over 5500 lines of code and a DNN-based controller [50].

The remainder of the paper is organized as follows. After summarizing related work (Sect. 2), the problem formulation for model-based safety verification of ACC is provided (Sect. 3). In Sect. 4, we present the verification framework based on set invariance. Then, we provide experimental results with several automated driving controllers (Sect. 5), followed by a discussion and conclusions (Sect. 6).

2 Related Work

While formal methods have been widely applied in the context of automated driving [30], the main focus so far has been on the behavior of an autonomous vehicle, rather than specific software code deployed in the vehicle. Formal verification of traditionally implemented controllers has been addressed, e.g., by utilizing model checking [29], counter-example-guided searching [44] or reachability analysis [2]. To avoid the need for verification, correct-by-design cruise control policies for longitudinal motion of platoons of autonomous vehicles have been synthesized using set invariance, which guarantee infinite-time collision avoidance [40]. As determining safe state sets is a computationally demanding task, online monitoring approaches have also been proposed as more scalable options

[22,26,36]. None of these methods perform safety verification or monitoring at code level, but rather use models of the code which might miss verification-relevant aspects of the implementation.

Many methods for input-output robustness certification of DNNs have also been proposed in recent years, including feed-forward multi-layer [18], deep feed-forward [24], and convolutional neural networks [14]. Formal proofs of closed-loop safety have also been obtained for DNN-based controllers and various system types, e.g., [8,20,21,28,39,41,45]; however, these methods rely on either approximations or abstraction of the to-be-verified controller or the system, and thus, tend to scale poorly with a growing complexity of the system. Properties for DNN-based perception components can also be verified using probabilistic analysis, e.g., for guiding airplanes on taxiways [35]. Satisfiability Modulo Theory (SMT) solvers have also been used for the automatic verification of DNNs with respect to safety properties in cyber-physical systems by using a dedicated interval constraint propagation [16] or by translating the closed-loop system into an SMT formula [42]. However, also in the context of DNN-based controllers safety verification has not been addressed for the deployed code.

Safety verification at code level has been addressed for an automated driving supervisor by automatically obtaining a finite discrete abstraction [32]. However, finite discrete abstraction approaches are not directly applicable to continuous controllers. The safety of adaptive cruise controller implementations was assessed by using temporal logic specifications embedded as monitors along with their execution, which can be checked using bounded model checking [33]. However, such methods are not guaranteed to terminate in a reasonable amount of time for every implementation. Instead of providing arguments for absolute correctness, the test coverage of automated driving functions, mostly provided by manual test drives and simulation runs in practice, can be extended by searching for specification counter-examples for the implementation utilizing reinforcement learning in simulation [11], by sampling initial conditions from the boundary of a controlled invariant set [6], or by (mostly) automatically extracting higher-level logic models from code [27], thus enabling exhaustive analysis for identifying potential errors prior to deployment. While these methods avoid human bias inherent to manual testing and can discover corner cases that may be overlooked otherwise, infinite automatic abstraction-based approaches are not guaranteed to scale well for all systems and cannot provide safety guarantees for the complete desired operation domain of the controller.

3 Problem Statement

We consider a common adaptive cruise controller system architecture, as for example used in [48] and already considered in [33], that is shown in Fig. 1. The driver activates or deactivates ACC, and provides a desired velocity v_d and a desired time headway t_{h_d}. The desired velocity is the target velocity for the automated driving vehicle (also referred to as the ego vehicle). The time headway is the amount of time after which the target object and the ego vehicle will collide,

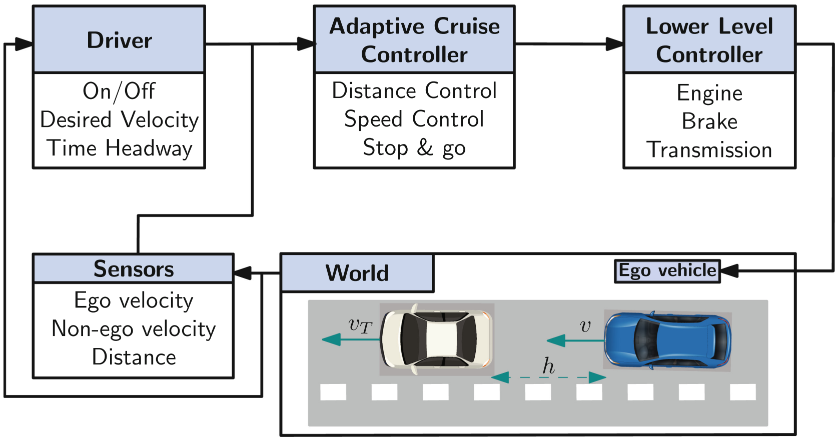


Fig. 1. A common longitudinal vehicle guidance architecture, adopted from [33]. The ACC provides a desired acceleration a to the lower level controllers, which convert it to engine, brake and transmission signals for the vehicle actuators.

given the current distance, or headway h, when the target object suddenly stops and the ego vehicle maintains its original velocity. Formally, the time headway t_h is the headway h over the current ego velocity v, i.e., $t_h = h/v$. The desired time headway t_{h_d} to the target object corresponds to the relative distance that eventually needs to be maintained. The information about the target object is measured by sensors such as radars, cameras or a lidar. This information is utilized in the adaptive cruise controller to produce the desired acceleration for reaching desired states for the ego vehicle. The acceleration commands from the ACC are used by the lower level controller such as the engine control unit and/or power train, the transmission controller and the brake controller.

Deriving an analytical specification suitable for formal verification is a challenging task for the complete chain of effects from the sensors all the way to the actuators of a vehicle. The focus of this work is on verifying the safe closed-loop behavior of ACC software implementations (Fig. 1). Therefore, perfect sensing and ideal lower level controllers are assumed. The analytical specification is based on a model of the vehicle's relevant longitudinal dynamics [37]. Non-linear force-balance equations are combined with exact feedback linearization to compensate non-linearities for the low level chain of effects [19]. Inspired by [34] $x = [v, v_T, h]^T$ is assumed as an overall state of the system and the linearized vehicle motion is described by $\dot{v} = a, \dot{v_T} = a_T$ and $\dot{h} = v_T - v$. These continuous differential equations are transformed into discrete time difference equations by exact discretization with an equidistant sampling time t_s. The continuous state variable x is replaced by the corresponding discrete time version x_t at a discrete time instant t. Further, a zero-order hold is used at a time instant t for the duration of t_s for a and a_T, which are denoted by a_t and $a_{T,t}$ in the discrete

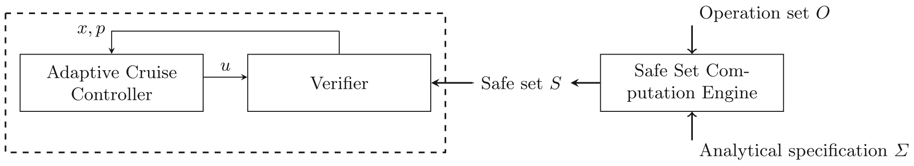

x, p
Adaptive Cruise Controller
u
Verifier
Safe set S
Operation set O
Safe Set Computation Engine
Analytical specification Σ

Fig. 2. Verification framework: based on the operation set O and the analytical specification Σ, the set S is used to check the safe closed-loop operation of ACC.

time domain. Thus, the assumed analytical specification is

$$\Sigma : x_{t+1}{=}Ax_t + Bu_t{=} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -t_s & t_s & 1 \end{bmatrix} x_t + \begin{bmatrix} t_s \\ 0 \\ -0.5t_s^2 \end{bmatrix} a_t + \begin{bmatrix} 0 \\ t_s \\ 0.5t_s^2 \end{bmatrix} a_{T,t}. \quad (1)$$

Without loss of generality, both the ego acceleration a and the target object acceleration a_T are bounded at all times, forming the set

$$O_u = \{a \in [a_{min}, a_{max}] \wedge a_T \in [a_{min}, a_{max}]\}. \quad (2)$$

In accordance with the relevant ISO Standard [1], the ACC computes a, so that either the ego vehicle velocity v reaches the driver desired velocity v_d, or so that the headway h to the target object driving with velocity v_T stays above a specified minimal value h_{min} and the current time headway stays above a specified minimal time headway $t_{h_{min}}$. If $v_d < h/t_{h_d}$, the target object is irrelevant, and the only safety requirement is given by (2). Since (2) can be guaranteed by a simple limiter, in this work we focus on the so called time gap or keep distance operation of ACC with a corresponding operational set

$$O_c = \{v_d \geq h/t_{h_d} \wedge h \geq h_{min} \wedge h/v \geq t_{h_{min}}\}. \quad (3)$$

Consider an implementation of the adaptive cruise controller (Fig. 1) in a general purpose programming language, e.g., C/C++, possibly containing a neural network, providing the acceleration a_t based on the state x_t of the analytical specification (1). The controller is assumed to be time-invariant and deterministic. We study how to verify that the controller implementation provides only control signals a_t for (1), such that their closed-loop operation always remains in (2) and (3) for all driver parameters v_d and t_{h_d}, and states x_t.

4 Framework

We propose a verification framework based on set invariance, as shown in Fig. 2. For a dynamical system, a set is invariant if every trajectory starting in this set remains in it for all times. For control systems, this means finding a control signal which is able to render a set invariant, i.e., a controlled invariant set. If all control signals produced by the controller yield states within the safe set, the controller can be certified as safe with respect to the analytical specification and the operation set. Thus, we compute the CIS with the analytical specification (1) for the operation sets (2) and (3). This allows us to obtain a corresponding safe set (denoted by the dashed area in Fig. 2) and effectively transform checking safety over (in)finite simulation traces over time into a set containment problem. Practically, set containment at code level is then accomplished by utilizing a state-of-the-art BMC. We first turn to obtaining the safe set.

4.1 Computing the Safe Set

The sets (2) and (3) can be readily encoded by means of the ego vehicle velocity v, the target object velocity v_T and the headway h, all contained in the state x. The operation set of the ACC is the union of (2) and (3) represented by a convex polytope over states and inputs, i.e.,

$$O = O_c \cup O_u. \tag{4}$$

Then, for (1) and (4) we compute the CIS (or an under-approximation thereof) as a polytope S_{CIS} represented by a finite number of inequalities N_S with corresponding matrices A_x^c and B_x^c, i.e., $S_{CIS} = \{x | A_x^c x \leq B_x^c\}, A_x^c \in \mathbb{R}^{N_S \times 3}, B_x^c \in \mathbb{R}^{N_S}$. Note that S_{CIS} is a subset of the operation set O, i.e., $S_{CIS} \subseteq O$. For any state $x_t \in S_{CIS}$ at time t, there exists at least one admissible control action a_t and target object acceleration $a_{T,t}$ with $[a_t, a_{T,t}]' \in O_u$, such that the following state x_{t+1} according to (1) remains within S_{CIS}. Thus, to verify that a controller is safe, for any state $x_t \in S_{CIS}$ we check that it produces an action, that yields a following state according to (1) within S_{CIS}. As (1) is linear and S_{CIS} is a polytope, the following safe state set S_{t+1} is also described by a polytope $S_{t+1} = \{(x, u) | A_x^c(Ax + Bu) \leq B_x^c\}$. Thus, the overall safe set comprises the union of the invariant set, the following safe state set and the admissible action region, i.e.,

$$S = S_{CIS} \cup S_{t+1} \cup O_u. \tag{5}$$

We use the method from [3] to compute a sequence of CISs with non-decreasing volume depending on a specified look-ahead time l. Figure 3 illustrates how the sets induced by each longer look-ahead time contain the ones induced by a shorter look-ahead time. This hierarchical relation allows to compute the CIS in closed-form in the original state space at the price of an increased number of inequalities. Note that our framework is compatible with any invariant set computation method which provides an under-approximation of the actual invariant set of the analytical specification.

4.2 Verification of Controller Implementations

A controller implementation can be deemed safe when it operates only within the bounded domain of the CIS for the analytical specification. Executions of a

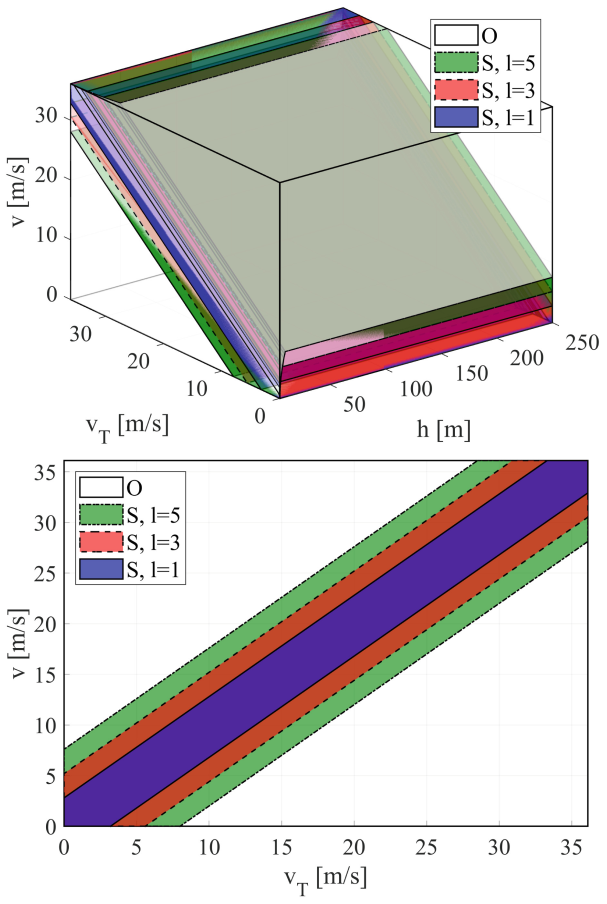


Fig. 3. CISs in 3D and 2D and volumes for look-ahead times $l = 1, V_1 = 5, 1.10^4$ (blue), $l = 3, V_3 = 8, 8.10^4$ (red), and $l = 5, V_5 = 1, 2.10^5$ (green). (Color figure online)

to-be-verified controller are checked against the safe set using a bounded model checker. By using a BMC the code is also implicitly checked for implementation and security flaws like integer overflows, out of bound array access, illegal pointer de-references etc., in addition to checking safety. We utilize a BMC to check if for any possible point in the safe set S, a control output is produced by the ACC under test that yields a following state, which is also inside of S. Since the driver parameters v_d and t_{h_d} may take integer values in their respective ranges, verification can be performed individually for each of the possible combinations in parallel. For simplicity, we consider a fixed pair of v_d and t_{h_d} in the following. The continuous variables x_t and $a_{T,t}$ are chosen freely within S by the BMC using `assume` statements. Based on these variables, the controller produces an output a_t, and the safety properties S are evaluated as a set containment check using an `assert` statement.

Due to the aforementioned hierarchical relation of the CIS computed with [3] for different lookahead times, a controller deemed safe for a lookahead time l is safe for all lookahead times l_1 with $1 \leq l_1 \leq l$. As higher look-ahead times l increase the size of the CIS they also increase the probability to discover counterexamples. Our approach also generalizes for maximal CIS, i.e., when there exists no higher lookahead time l with a larger corresponding CIS. In fact, as (1) is a discrete-time linear system without disturbances, the maximal controlled invariant set, which contains all possible safe scenarios, can be approximated well (e.g. using [17]), at the price of higher complexity of S_{CIS}. Since most CIS computation approaches provide an under-estimation of the actual CIS, the proposed verification procedure is sound. If the computed CIS is exact, the procedure is also complete.

4.3 Verification Decomposition for Large Neural Network Controllers

As BMCs enumerate possible branches during state space exploration, verification for large neural networks is not guaranteed to terminate in a reasonable amount of time. To overcome this problem, we use a three-step decomposition approach. First, states are selected based on a heuristic criterion which ensures that a representative set within the operational domain is tested for package deployment. For this finite set of individual states x_t in S, the controller output is checked to remain in S. This provides assurance that the input-output behavior of the overall controller, including the supporting code, performs as expected before doing the DNN verification.

Next, we take inspiration from recent research [9,10] that proposes to verify a large DNN by initially reducing it to a simpler, smaller DNN for verification (abstraction), and iteratively making it more complex as needed (refinement). Therefore, we replace the original DNN with a simpler DNN with the same inputs and outputs, and use BMC to check that all code required for operational deployment (DNN model loading, DNN inference, etc.) works as expected. For our purposes, we assume that an abstraction approach was used to obtain the smaller DNN, and that the supporting code needed for the original DNN is being fully executed for the deployment of the smaller DNN. This ensures that the supporting code is checked for implementation flaws like integer overflows, out of bound access etc. In this case, the operation is not required to always remain in S, since the smaller DNN model might not fulfil the safety specification.

As a final step, we use an off-the-shelf DNN verifier to check if, for any point in S, a control a_t is produced by the original DNN that together with (1) yields a following state in S. Only if the verification is successful in all three steps, the safety specification is considered as verified.

5 Experiments

We applied the proposed verification framework to the following four common classes of adaptive cruise controllers, which are widely used for automated driving both in simulation environments and in industrial applications:

1. A switching proportional controller (PC) with the gain $k_P = 3$ and
$$a_t = k_P(v_t - \min(v_d, h_t/t_{h_d})),$$
where the min-part takes care of the time gap and adapt speed modes.
2. A Nonlinear Controller (NC) known as the Intelligent Driver Model [47]:
$$a_t = a_{max}\left(1 - \left(\frac{v_t}{v_d}\right)^{\delta} - \left(\frac{d(x_t)}{d_{T,t}}\right)^2\right),$$
$$d(x_t) = h_t + v_t t_{h_d} + \frac{v_t(v_t - v_{T,t})}{2\sqrt{a_{max}a_{com}}},$$
where $d_{T,t} = 1.8\tilde{v}_t$ is the desired distance between the two vehicles, which is around half of the current ego vehicle's velocity $\tilde{v}_t$ in km/h (the recommended minimum distance according to German traffic rules) and $a_{com} = 1.5\, m/s^2$ is the absolute value of the comfortable acceleration.
3. A Model Predictive Controller (MPC) [31] using the model (1). The target object keeps $a_T = 0$ throughout the optimization horizon with $N = 5$ samples. With the initial state $x_{\tilde{t}}|_{\tilde{t}=0}$, the following quadratic program is solved at each state x_t:
$$\begin{aligned} &\min_{a_{\tilde{t}}} \sum_{\tilde{t}=0}^{N}(\|v_{\tilde{t}} - \min(v_d, h_{\tilde{t}}/t_{h_d})\| + \|a_{\tilde{t}}\|), \\ &\text{s.t. } \forall\tilde{t} \in [0, N], (1); \\ &\quad\;\; \forall\tilde{t} \in [1, N], a_{T,\tilde{t}} = 0; a_{\tilde{t}} \in O_u; x_{\tilde{t}} \in O; \\ &\quad\;\; x_{\tilde{t}}|_{\tilde{t}=0} = x_t. \end{aligned}$$
The controller is implemented using the Multi-Parametric Toolbox [17]. An explicit solution comprising 348 state feedback controllers over the relevant ODD space is exported to C.
4. A Neural Network Controller (NNC) [50], which combines imitation learning of recorded demonstrations and optimizing a reward function incorporating safety, efficiency, and comfort metrics to maximize cumulative rewards through simulation trials. Deep deterministic policy gradient (DDPG) is utilized to learn an actor network together with a critic network. We focus on verifying the actor with an input x_t and an output a_t. The actor has one hidden layer with 30 neurons. For all layers, the rectified linear unit activation function was used.

Table 1. Controller falsification/verification times in $[min]/[min]$ for controllers, for different look-ahead times l and safe sets S with corresponding number of inequalities N_S.

controller	code lines	Look-ahead l / N_S			
		1/46	2/50	3/54	4/58
PC	10	1.1/–	1.5/–	2.0/–	2.5/–
NC	15	–/1.5	2.1/–	3.7/–	4.9/–
MPC	5521	–/1.7	–/2.2	–/4.0	–/5.4
NNC	1672	timeout	timeout	timeout	timeout
NNC*	1672	–/13.5	16.4/–	20.8/–	30.1/–

'-' on the left implies that no counter-example was found
'-' on the right implies that verification was unsuccessful (a counter-example was found)
NNC denotes the times using BMC only for verification
NNC* represents the cumulative times for verification based on decomposition and BMC

5.1 Setup

The sampling time $t_s = 0.2\,\mathrm{s}$ is chosen for (1). The parameters $a_{max} = 2\,\mathrm{m/s}^2$, $a_{min} = -4\,\mathrm{m/s}^2$, $v_t, v_{T,t} \in (0, 130]\,\mathrm{km/h}$, $h_{max} = 250\,\mathrm{m}$, $h_{min} = 5\,\mathrm{m}$, $t_{h_{min}} = 1\,\mathrm{s}$ are used. A desired ego velocity $v_d = 130\,km/h$ and a desired time headway $t_{h_d} = 1.8\,\mathrm{s}$ are assumed in the case study. CBMC 5.95.1 [7] is utilized as a bounded model checker with a timeout of 1 h. The analysis was performed on a standard workstation with an Intel Core i7-11850H CPU with 64 GB DDR4 RAM. Regarding the decomposition approach for neural network controllers, we first check the actual DNN for the individual states $x_t \in \{[0, 0, 0], [15, 5, 5]\}$. Then, the supporting code is checked using the BMC with a simplified neural network with the same inputs and outputs, but only 3 neurons in the hidden layer. Finally, using the Marabou [25] DNN verifier we check the actual DNN.

5.2 Experimental Results

Computing the CIS for $l = 4$ took 1 min and computing the CIS for all $l < 4$ took less than half a minute. As this procedure is required to be executed only once and the resulting invariant sets are reused for evaluating the safety of all controller types, we focus on the runtime for verifying the code in the following. Table 1 shows results from the development-time verification of the controllers. As the complexity of S_{CIS} increases with higher l, the verification times increase accordingly. As an increasing l increases the volume of S_{CIS}, the probability to discover a counter-example also grows with higher l. Therefore, it is not surprising that NC could be verified for S_c with $l = 1$, but falsified with $l \geq 2$. The MPC, which was designed to consider the analytical specification and operation set, was verified in all cases. Note that simply using BMC on the NNC timed out for all invariant sets. A similar result was reported in [42] for small DNNs controlling a cart-pole system, presumably caused by the non-linearity and non-invertability of the DNN. However, using our proposed decomposition, we were able to falsify the NNC for all invariant sets. Using BMC, checking the supporting code was done in 10.5 min for the auxiliary neural network, and in approx. 1 min for each individual state for the actual DNN. An extra minute was needed for the Marabou verifier to check the actual DNN.

All falsifying input-output pairs denote an insufficient ego vehicle deceleration, while the target object decelerates with a_{min} and the time headway is not large enough. While the PC decelerates slightly in this case, the NC and the NNC decelerate with nearly a_{min}. For all falsified controllers, either algorithmic improvements are needed, or a dedicated supervisor component has to be introduced to guarantee safety at the price of some performance loss.

5.3 Discussion

The experimental results presented in the previous section show that our approach has several benefits. The considered ACCs were used to control traffic agents in a simulation environment. Using our method we could achieve more realistic and safe closed-loop behavior. First, we were able to find safety flaws at code level for all controllers except for the MPC (Table 1), which may have remained uncovered upon simulation- and field-based testing only. Second, the acquired falsifying samples correspond to driving scenarios that provide feedback for improving the considered ACC controllers – either by deriving additional automated tests or by revisiting algorithmic solutions. Further, applying formal verification even only to some software modules greatly supports the overall system-level analysis and design, e.g., by providing hints where a dedicated supervisor component might be required as a safety assurance measure. Third, for neural network controllers, our approach allows falsifying examples to be used to indicate possible gaps in the collected training data or undesirable biases in the current training stage. Finally, our framework can be integrated into the verification and validation process within the development of automated driving functions. As the only manual modeling step is related to deriving the analytical specification and verifying controller implementations is possible in an automated manner within minutes, our solution can be included in the continuous integration (CI) process, where it is connected to consecutive versions of the to-be-deployed controller software.

As falsification or verification was possible in all considered cases, the proposed framework is expected to be suitable for various controller types. The method is not limited to the analytical specification (1). While this paper uses a linear system for which computing a CIS is known to have polynomial complexity [38], the presented verification approach can be readily used for analytical specifications and operation sets, which allow computing a CIS. Obtaining the CIS is possible for many nonlinear systems, e.g., [12]. Even though our work focuses on ACC, its key ideas can be transferred for other cyber-physical systems.

While the presented scheme has great potential for automated safety verification of many safety-critical controllers, several limiting factors have to be mentioned.

First, the obtained verification result does not generalize for all possible real world driving scenarios and systems. Our framework provides either a proof that the controller satisfies the analytical specification or a proof of non-compliance accompanied by a counterexample. These proofs pertain to the correctness of the controller's behavior when applied to real-world driving. However, full confidence in these statements necessitates ensuring that the analytical specification accurately reflects real-world traffic dynamics, physics, and perception/actuation mechanisms. Over-estimating possible violations of the analytical specification is preferred to under-estimating them to accurately reflect error-freeness, with the additional check of counterexamples in the actual automated driving system to eliminate false positives. While the current approach aims to over-estimate violations, further investigation is needed to understand the extent of its accuracy and the implications for proving correctness at system level.

Second, defining the operation set and the parameter set for model checking might become challenging with an increasing complexity of the analytical specification and the to-be-verified controller. Obtaining a suitable analytical specification for controller verification requires a trade-off between precision and complexity. Special hybrid systems formulations amenable for verification might be a good choice, e.g., [49].

Third, using significantly more complex dynamical system models as analytical specifications might make the computation of the CIS infeasible or the bounded model checking intractable. Even when linear models are employed, the choice of l affects the computational effort of the method [3] and poses a trade-off with respect to the size of the operational domain for verification. In general, computing controlled invariant sets for nonlinear systems is challenging. Some works employ convex approximations to reduce the computational complexity [12]. Other works exploit structural properties, e.g., for polynomial systems, to compute exact sets [4]. However, as the maximal controlled invariant set of a nonlinear system is typically non-convex, in general, the obtained CIS can be conservative. A closely related question for future exploration is whether other desired properties of the closed-loop operation of the controllers can be captured by invariants and consequently verified using our approach. For instance, in camera-based systems, a closed-loop operation objective would be to minimize unnecessary fluctuations of the control signals when the images remain stable.

Forth, the particular implementation plays a deciding role for the verification complexity. As reported in [33], even using certain C++ Standard Library functions might not be the best choice for verification. Finally, some bounded model checkers might be more suitable for verifying a particular piece of code than others. Similarly, different DNN verifiers might perform better than others for particular DNNs.

Finally, while the proposed framework provides functional safety robustness guarantees with respect to the analytical specification, we note that some DNN verifiers are not guaranteed to provide guarantees against all possible malicious network inputs and/or network architectures. For example, floating point errors have been observed to occasionally cause incorrect results with some DNN verifiers on large scale benchmarks [23]. Therefore, it is advisable to study the specific features of the used tools in detail.

6 Conclusions

We proposed an automatic safety verification approach for adaptive cruise controllers for automated driving vehicles at code level, and applied it to both traditional and neural-network-based controllers. By computing a controlled invariant set for a given analytical specification, our approach allows obtaining a safe set for the closed-loop operation, therefore enabling the verification of controller implementations by utilizing a bounded model checker. Furthermore, by proposing a three-step verification decomposition, we were able to verify a neural-network-based controller, for which off-the-shelf bounded model checkers timed out. The experimental results confirm that both traditionally implemented and neural-network-based adaptive cruise controllers can be verified offline within a time frame of minutes on a regular computer, thus emphasizing the low computational overheads of the framework for cyber-physical system controllers.

In future work, we plan to apply our approach to the verification of additional types of controllers from the automotive domain, e.g. those responsible for automated lane keeping or lane changing. In addition, we intend to extend the approach to consider uncertainties in the system model, e.g., by using probabilistic [5,15] and/or statistical model checking.

Acknowledgments. This work was partly supported by the Assuring Autonomy International Programme, a partnership between Lloyd's Register Foundation and the University of York, UK.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Intelligent transport systems - adaptive cruise control systems - performance requirements and test procedures (2018). ISO Standard 15622:2018
2. Alam, A., Gattami, A., Johansson, K.H., Tomlin, C.J.: Guaranteeing safety for heavy duty vehicle platooning: safe set computations and experimental evaluations. Control. Eng. Pract. **24**, 33–41 (2014). https://doi.org/10.1016/j.conengprac.2013.11.003
3. Anevlavis, T., Liu, Z., Ozay, N., Tabuada, P.: Controlled invariant sets: implicit closed-form representations and applications. IEEE Trans. Autom. Control, pp. 1–16 (2023). https://doi.org/10.1109/TAC.2023.3336819
4. Ben Sassi, M.A., Girard, A.: Computation of polytopic invariants for polynomial dynamical systems using linear programming. Automatica **48**(12), 3114–3121 (2012). https://doi.org/10.1016/j.automatica.2012.08.014

5. Calinescu, R., Češka, M., Gerasimou, S., Kwiatkowska, M., Paoletti, N.: Efficient synthesis of robust models for stochastic systems. J. Syst. Softw. **143**, 140–158 (2018)
6. Chou, G., Sahin, Y.E., Yang, L., Rutledge, K.J., Nilsson, P., Ozay, N.: Using control synthesis to generate corner cases: a case study on autonomous driving. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(11), 2906–2917 (2018). https://doi.org/10.1109/TCAD.2018.2858464
7. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
8. Dawson, C., Gao, S., Fan, C.: Safe control with learned certificates: a survey of neural lyapunov, barrier, and contraction methods for robotics and control. IEEE Trans. Rob. **39**(3), 1749–1767 (2023). https://doi.org/10.1109/TRO.2022.3232542
9. Elboher, Y.Y., Cohen, E., Katz, G.: On applying residual reasoning within neural network verification. Softw. Syst. Model. pp. 1–16 (2023). https://doi.org/10.1007/s10270-023-01138-w
10. Elboher, Y.Y., Gottschlich, J., Katz, G.: An abstraction-based framework for neural network verification. In: Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32, pp. 43–65. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_3
11. Favrin, A., Nenchev, V., Cenedese, A.: Learning to falsify automated driving vehicles with prior knowledge. IFAC-PapersOnLine (2020). https://doi.org/10.1016/j.ifacol.2020.12.2036, iFAC World Congress 2020 (IFAC'2020), Berlin
12. Fiacchini, M., Alamo, T., Camacho, E.: On the computation of convex robust control invariant sets for nonlinear systems. Automatica **46**(8), 1334–1338 (2010). https://doi.org/10.1016/j.automatica.2010.05.007
13. Garrido, F., Resende, P.: Review of decision-making and planning approaches in automated driving. IEEE Access **10**, 100348–100366 (2022). https://doi.org/10.1109/ACCESS.2022.3207759
14. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 3–18 (2018). https://doi.org/10.1109/SP.2018.00058
15. Gerasimou, S., Cámara, J., Calinescu, R., Alasmari, N., Alhwikem, F., Fang, X.: Evolutionary-guided synthesis of verified pareto-optimal MDP policies. In: 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 842–853. IEEE (2021)
16. Grundt, D., Jurj, S.L., Hagemann, W., Kröger, P., Fränzle, M.: Verification of sigmoidal artificial neural networks using ISAT. In: International Workshop on Symbolic-Numeric methods for Reasoning about CPS and IoT (2022). https://doi.org/10.4204/EPTCS.361.6
17. Herceg, M., Kvasnica, M., Jones, C.N., Morari, M.: Multi-parametric toolbox 3.0. In: European Control Conference (ECC), pp. 502–510 (2013). https://doi.org/10.23919/ECC.2013.6669862
18. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification, pp. 3–29. Springer International Publishing, Cham (2017)
19. Ioannou, P., Chien, C.: Autonomous intelligent cruise control. IEEE Trans. Veh. Technol. **42**(4), 657–672 (1993). https://doi.org/10.1109/25.260745

20. Ivanov, R., Carpenter, T., Weimer, J., Alur, R., Pappas, G., Lee, I.: Verisig 2.0: verification of neural network controllers using Taylor model preconditioning. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification, pp. 249–262. Springer International Publishing, Cham (2021)
21. Ivanov, R., Carpenter, T.J., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verifying the safety of autonomous systems with neural network controllers. ACM Trans. Embed. Comput. Syst. **20**(1) (2020). https://doi.org/10.1145/3419742
22. Jacumet, R., Rathgeber, C., Nenchev, V.: Analytical safety bounds for trajectory following controllers in autonomous vehicles. In: Proceedings of International Conference on Control, Decision and Information Technologies (CoDIT) (2023). https://doi.org/10.1109/CoDIT58514.2023.10284507
23. Jia, K., Rinard, M.: Exploiting verified neural networks via floating point numerical error. In: Drăgoi, C., Mukherjee, S., Namjoshi, K. (eds.) Int. Static Analysis Symposium. pp. 191–205. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-88806-0_9
24. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kuncak, V. (eds.) Proceedings of the 29th International Conference on Computer Aided Verification (CAV '17). Lecture Notes in Computer Science, vol. 10426, pp. 97–117. Springer, heidelberg, Germany (2017)
25. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification, pp. 443–452. Springer International Publishing, Cham (2019)
26. Kojchev, S., Klintberg, E., Fredriksson, J.: A safety monitoring concept for fully automated driving. In: 2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC), pp. 1–7 (2020). https://doi.org/10.1109/ITSC45102.2020.9294307
27. König, L., et al.: Towards safe autonomous driving: model checking a behavior planner during development. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 44–65. Springer Nature Switzerland, Cham (2024)
28. Lopez, D.M., Choi, S.W., Tran, H.D., Johnson, T.T.: NNV 2.0: the neural network verification tool. In: Enea, C., Lal, A. (eds.) Computer Aided Verification, pp. 397–412. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-37703-7_19
29. Lygeros, J., Godbole, D.N., Sastry, S.: A verified hybrid controller for automated vehicles. In: Proceedings of 35th IEEE Conference on Decision and Control vol. 2, pp. 2289–2294 (1996)
30. Mehdipour, N., Althoff, M., Tebbens, R.D., Belta, C.: Formal methods to comply with rules of the road in autonomous driving: state of the art and grand challenges. Automatica **152**, 110692 (2023). https://doi.org/10.1016/j.automatica.2022.110692
31. Naus, G., Ploeg, J., Van de Molengraft, M., Heemels, W., Steinbuch, M.: Design and implementation of parameterized adaptive cruise control: an explicit model predictive control approach. Control. Eng. Pract. **18**(8), 882–892 (2010). https://doi.org/10.1016/j.conengprac.2010.03.012
32. Nenchev, V.: Automated behavior modeling for verifying safety-relevant modules. In: Proceedings of IEEE International Conference on Robotic Computing (IRC) (2021). https://doi.org/10.1109/IRC52146.2021.00021
33. Nenchev, V.: Model checking embedded adaptive cruise controllers. Robot. Auton. Syst. **167**, 104488 (2023). https://doi.org/10.1016/j.robot.2023.104488

34. Nilsson, P., et al.: Correct-by-construction adaptive cruise control: two approaches. IEEE Trans. Control Syst. Technol. **24**(4), 1294–1307 (2016). https://doi.org/10.1109/TCST.2015.2501351
35. Păsăreanu, C.S., et al.: Closed-loop analysis of vision-based autonomous systems: a case study. In: Enea, C., Lal, A. (eds.) Computer Aided Verification, pp. 289–303. Springer Nature Switzerland, Cham (2023)
36. Pek, C., Manzinger, S., Koschi, M., Althoff, M.: Using online verification to prevent autonomous vehicles from causing accidents. Nat. Mach. Intell. **2**(9) (2020). https://doi.org/10.1038/s42256-020-0225-y
37. Rajamani, R.: Vehicle Dynamics and Control. Mechanical Engineering Series, Springer, US (2011)
38. Raković, S., Kerrigan, E., Mayne, D., Kouramas, K.: Optimized robust control invariance for linear discrete-time systems: theoretical foundations. Automatica **43**(5), 831–841 (2007). https://doi.org/10.1016/j.automatica.2006.11.006
39. Ruoss, A., Baader, M., Balunović, M., Vechev, M.: Efficient certification of spatial robustness. In: Thirty-Fifth AAAI Conference on Artificial Intelligence (2021)
40. Sadraddini, S., Sivaranjani, S., Gupta, V., Belta, C.: Provably safe cruise control of vehicular platoons. IEEE Control Syst. Lett. **1**(2), 262–267 (2017). https://doi.org/10.1109/LCSYS.2017.2713772
41. Santa Cruz, U., Shoukry, Y.: Nnlander-verif: a neural network formal verification framework for vision-based autonomous aircraft landing. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods, pp. 213–230. Springer International Publishing, Cham (2022)
42. Scheibler, K., Winterer, L., Wimmer, R., Becker, B.: Towards verification of artificial neural networks. In: Heinkel, U., Rößler, M., Kriesten, D. (eds.) Proceedings of the 18th Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen" (MBMV), pp. 30–40. Technische Universität Chemnitz, Germany, Chemnitz, Germany (2015)
43. Stern, R., Gunter, G., Work, D.B.: Modeling and assessing adaptive cruise control stability: experimental insights. In: 2019 6th International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS), pp. 1–8 (2019). https://doi.org/10.1109/MTITS.2019.8883330
44. Stursberg, O., Fehnker, A., Han, Z., Krogh, B.H.: Verification of a cruise control system using counterexample-guided search. Control. Eng. Pract. **12**, 1269–1278 (2004)
45. Sun, X., Khedr, H., Shoukry, Y.: Formal verification of neural network controlled autonomous systems. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 147–156. HSCC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3302504.3311802
46. Tian, Y., Pei, K., Jana, S., Ray, B.: Deeptest: automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th International Conference on Software Engineering, pp. 303–314. ICSE '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3180155.3180220
47. Treiber, M., Hennecke, A., Helbing, D.: Congested traffic states in empirical observations and microscopic simulations. Phys. Rev. E **62**, 1805–1824 (2000)
48. Widmann, G.R., et al.: Comparison of lidar-based and radar-based adaptive cruise control systems. SAE Trans. **109**, 126–139 (2000)

49. Wongpiromsarn, T., Mitra, S., Lamperski, A., Murray, R.M.: Verification of periodically controlled hybrid systems: application to an autonomous vehicle. ACM Trans. Embed. Comput. Syst. **11**(S2) (2012). https://doi.org/10.1145/2331147.2331163
50. Zhu, M., Wang, Y., Pu, Z., Hu, J., Wang, X., Ke, R.: Safe, efficient, and comfortable velocity control based on reinforcement learning for autonomous driving. Trans. Res. Part C: Emerg. Technol. **117**, 102662 (2020). https://doi.org/10.1016/j.trc.2020.102662

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

A Case Study on Formal Equivalence Verification Between a C/C++ Model and Its RTL Design

Gaetano Raia[1], Gianluca Rigano[2], David Vincenzoni[2](✉), and Maurizio Martina[1]

[1] Politecnico di Torino, Torino (TO), Italy
gaetanomaria.raia@studenti.polito.it, maurizio.martina@polito.it
[2] STMicroelectronics, Agrate Brianza (MB), Italy
{gianluca.rigano,david.vincenzoni}@st.com

Abstract. In the field of communication system products, most datapath Digital Signal Processing algorithms are initially developed at a high-level in MATLAB® or C/C++. Subsequently, design engineers use these models as a reference for implementing Register Transfer Level designs. The conventional approach to verify their equivalence involves extensive Universal Verification Methodology dynamic simulations, which can last for months and require significant verification efforts. However, some elusive errors might still occur because it is infeasible to explore all input combinations with this method. On the other hand, Formal Equivalence Verification aims to verify that a Register Transfer Level design is functionally equivalent to the reference high-level C/C++ model across all possible legal states. With recent advancements in formal solver technology, Formal Equivalence Verification provides a distinct benefit by using mathematical methods to ensure that the Register Transfer Level (timed) matches the original high-level C/C++ model (untimed). This drastically reduces the verification time and ensures the exhaustive coverage of the design state space. This paper presents an in-depth exploration of complex Finite State Machine with datapath verification, specifically focusing on Multiplier-Accumulator, Tone Generator, and Automatic Gain Control, by employing the formal equivalence methodology. Although these signal processing blocks were previously verified throughout Universal Verification Methodology dynamic simulations, Formal Equivalence Verification was able to identify hard-to-find bugs in just a few weeks by utilizing the new workflow, thereby streamlining the verification process.

Keywords: Formal Equivalence Verification · Jasper™ C2RTL App · C/C++ model · RTL design · Formal datapath verification

1 Introduction

Integrated circuits have become a cornerstone in both commercial and industrial domains. As the demand for more sophisticated electronic devices has surged,

© The Author(s) 2025

the intricacy of these devices has considerably increased. This has necessitated continuous evolution in design methodologies and verification processes to meet the advancing technological requirements, as the cost of finding and solving bugs has grown exponentially throughout the design process [14,16].

Verification is a critical process designed to confirm that a Device Under Verification (DUV) maintains its intended behavior throughout its implementation. In the domain of System-on-a-Chip, a variety of verification technologies has been established. These technologies are essential for ensuring the functionality of these devices, which are complex designs integrating multiple disciplines. While various categories of design flaws contribute to integrated circuits re-spins, functional flaws remain the leading cause of bugs [18].

Additionally, the median percentage of total integrated circuit project time dedicated to functional verification is approximately between 50% and 60% [17]. This figure can vary depending on the design; projects that utilize existing pre-verified Intellectual Property (IP) may require less verification time, whereas those with newly developed IP could require more. In general, test planning and testbench development are the areas in which verification engineers spend the most of their time, respectively 47% and 21% [17].

Over the last decade, functional verification has been mainly conducted through the use of Universal Verification Methodology (UVM) testbench environments. A testbench serves as a verification framework that administers a predefined set of input patterns, also referred to as stimuli, to the Register Transfer Level (RTL) design. The primary function of a testbench is to facilitate the observation of whether the DUV yields the correct outputs, compared to a reference model, in reaction to these stimuli, as shown in Fig. 1.

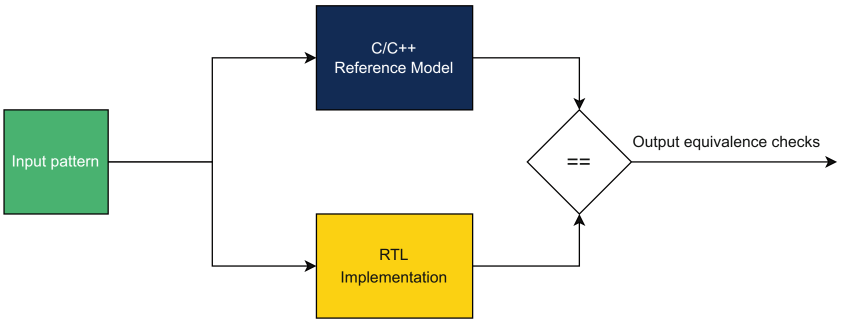


Fig. 1. Conceptual representation of functional equivalence verification between a C/C++ reference model and an RTL implementation

However, specifying millions of test vectors for exhaustive verification becomes impractical in simulation-based approaches due to the exponential increase in scenarios with the number of input bits: for instance, a 32×32 bit integer multiplier would have 2^{64} total input combinations, at one million combinations checked per second, resulting into hundreds of thousands of processor years. Random simulations are a practical alternative, providing a statistical overview of compliance with specifications rather than exhaustive verification. Yet, purely random inputs can miss corner cases or produce unrealistic scenarios. Constrained-random simulations address this by guiding random input generation within defined parameters, improving coverage but still not guaranteeing full design space exploration.

Moreover, this approach is resource-intensive due to the number of components involved and is likely prone to subtle errors during the construction of the verification environment. Ultimately, the UVM requires considerable effort to verify its correctness prior to initiating the verification process. This results in prolonged verification cycles, which may prove sub-optimal for projects facing strict time-to-market constraints or operating with limited resources.

As electronic designs have become more complex and the time allocated for design cycles has decreased, the industry has developed a suite of verification methodologies and Electronic Design Automation (EDA) tools to address these challenges. This paper introduces an innovative verification flow that leverages Formal Equivalence Verification (FEV) to check that RTL designs match C/C++ models. This approach, based on mathematical properties, ensures exhaustive coverage and a significant reduction in verification time when compared to traditional UVM dynamic simulations. The comprehensive version of this paper, which includes in-depth discussions of the validated designs and the methodologies employed, is available for reference [15].

2 Formal Equivalence Checking in C-vs-RTL Scenarios

Functional verification is an essential step in the design process, aimed at confirming that the implementation reflects the design intent. The reconvergence model [2] suggests that the purpose of verification consists in ensuring that the result of some transformation, such as RTL coding, is as expected. This can be accomplished through a secondary path reconverging with the primary design path at a shared origin, namely the specification model (see Fig. 2).

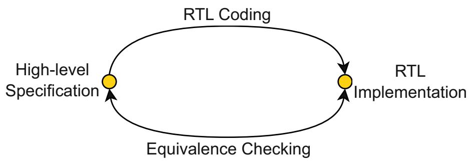


Fig. 2. Recovergence model of functional verification through equivalence checking

Formal equivalence checking employs mathematical reasoning to confirm that an *implementation* adheres to a *specification*. Formal verification leverages a language with precisely defined syntax and semantics to encapsulate the system's intended behavior, utilizing the IEEE standard for SystemVerilog Assertion (SVA) [8]. Through mathematical proofs, formal verification ensures the correctness of the Device Under Verification regardless of the input values, as it implicitly consider any legal case in the design state space. Two models are considered equivalent if, upon exhaustive analysis of all possible cases, the formal verification tool has not identified any discrepancies - commonly referred to as counterexamples - that would negate the equivalence.

To elucidate the mechanisms employed in today's equivalence-checking tools, it is instructive to consider a common computational equivalence model known as *miter*. This model effectively acts as a product machine that combines two Finite State Machine (FSM) designs by aligning each corresponding pair of primary inputs and connecting each pair of outputs to an XOR gate, as shown in Fig. 3. Establishing equivalence between two machines, denoted as $M_{\text{spec}}(X)$ for the specification machine and $M_{\text{impl}}(X)$ for the implementation machine, necessitates the demonstration that for any given input sequence $X = (x_1, x_2, \ldots, x_n)$, the outputs of the product machine consistently yield a zero value. Equivalence is thus confirmed by proving the nonsatisfiability of Eq. 1 across all possible inputs X, where $\oplus$ denotes the XOR gate operation.

$$M_{spec}(X) \oplus M_{imp}(X) \tag{1}$$

While there are various methods to address this challenge, such as Binary Decision Diagrams (BDDs) and Satisfiability (SAT) algorithms [14,16], they are beyond the scope of this paper.

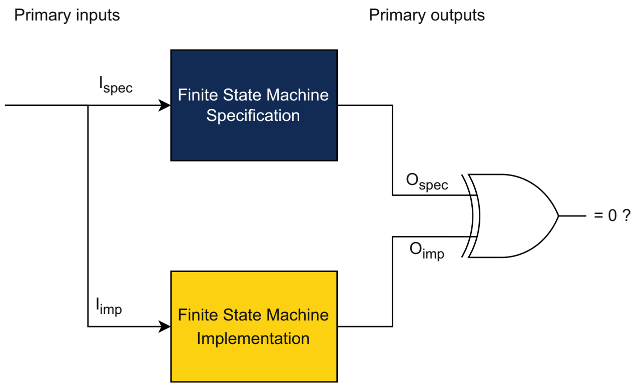

Primary inputs
Primary outputs
I_spec
Finite State Machine Specification
O_spec
= 0 ?
O_imp
I_imp
Finite State Machine Implementation

Fig. 3. Miter model of two FSM designs to verify through formal equivalence

2.1 Jasper$^{\text{TM}}$ C2RTL App

The advent of a novel category of formal engines embedded in Cadence$^{\circledR}$ Jasper$^{\text{TM}}$ C2RTL App, specifically optimized for evaluating RTL datapath implementations against their C/C++ algorithmic specifications, has marked a significant leap in verification performance. These specialized engines are now capable of delivering performance that is up to 100 times more efficient than that of traditional general-purpose formal engines [10]. This innovation represents a substantial breakthrough for semiconductor companies, which frequently depend on robust, standardized EDA tools to manage the complexities inherent to design processes.

The integration of early-design formal verification checks into the design cycle can dramatically enhance the efficiency and effectiveness of the verification efforts. Nevertheless, it is essential to demonstrate that C/C++ models accurately capture the design intent, as these models often serve as the starting point for computational block development due to their abstraction capabilities, simulation speed, verification efficiency, and standard usage in the semiconductor industry. High-level C/C++ models can be easily verified at system level compared to RTL, to understand if they fulfill with their specifications: if so, they become the *golden reference* for the related RTL implementation. Implementing a redundancy layer enhances verification reliability by pinpointing whether inconsistencies stem from RTL coding mistakes or inaccuracies in translating design intent into C/C++, thereby preserving design integrity.

For the sake of clarity and focus, this paper does not delve into the specifics of formal engines, as their intricate details fall outside the scope of the current discussion (refer to [3,4] for any insight). The emphasis here is on the broader implications of these advanced tools and their impact on the semiconductor industry's verification practices, rather than on the technical nuances of the engines themselves.

3 The Verification Flow

In formal verification, intended behaviors are encapsulated as *properties*, which represent collections of logical and temporal relationships among subordinate Boolean and sequential expressions, usually written in SVA language. Over the past decades, verification engineers have been compelled to develop extensive sets of properties to capture all conceivable behaviors for verification. This approach has been both time-consuming and prone to risk, as the potential for overlooking certain properties could lead to incomplete verification and undetected design errors. The pivotal advantage of FEV in C-vs-RTL scenarios lies in the automatic comparison of the two models facilitated by the automatic generation of *assertions*. Concisely, an assertion is a declarative statement that specifies a property which must always hold. This automation streamlines the verification process, significantly reducing the manual effort and the associated risk of human error in property specification. Within this context, the formal tool possesses the capability to generate mathematical properties checking that both the designs produce identical outputs under the same input conditions.

Although the verification methodology enhances autonomy in property generation and checking, it is not fully independent and continues to necessitate human guidance for configuring the verification environment, delineating the state space of the design, and addressing convergence issues that arise from state-space explosion in intricate digital circuits.

Despite these challenges, Jasper^{TM} C2RTL App is able to handle a large variety of datapath algorithms, such as unit arithmetic operations, high-level image processing algorithms, and encryption/decryption models [13]: in addition, it handles pipelines, feedback loops, floating-point and more [10]. The following list describes the innovative verification flow to apply FEV in verifying digital circuits, without the need to develop verification components and test vectors:

1. **C/C++ Model Compilation**: the tool adheres to the latest ANSI C++ standards and integrates with prevalent math libraries [10].
2. **RTL Compilation**: the tool supports SystemVerilog RTL implementations [10]. For non-SystemVerilog Hardware Description Language (HDL), equivalence checking tools ensure consistency with the original design [5,11].
3. **I/O Port Mapping**: verification engineers map input and output ports between the specification and implementation.
4. **Clock and Reset Definition**: the clock signal is identified in the RTL, and reset signal polarity is specified to detect the reset state.
5. **Input Assumptions**: engineers define input signal dynamics and protocols, acting as constraints to exclude illegal behaviors and prevent spurious counterexamples.
6. **Formal Engine Configuration**: while engineers can select optimizations for datapath-specific issues, leveraging the tool's machine learning-based configuration may yield optimal results [13], especially at the beginning.
7. **Coverage Property Specification**: engineers outline coverage properties to evaluate the DUV in targeted scenarios. These are employed by the user to prove the existence of at least one legal case fulfilling a specific condition, thereby facilitating the identification of the most concise path satisfying it.
8. **Proof Execution**: the tool checks for discrepancies between models using automatically generated and manually written assertions.

For the sake of clarity, Table 1 outlines all possible outcomes when proving an assertion. If the verification runs extensively without finding bugs, the verification user may decide to conclude the process, especially under tight design cycle deadlines. Generally, if a proof runs for more than 24 h without a result, it may be necessary to rewrite or decompose the proof or to try different engine modes [4]. This paper endeavors to delineate the challenges associated with managing sophisticated real-world digital circuits developed by STMicroelectronics and to outline effective strategies for ensuring convergence. To this end, it is advisable to construct a comprehensive verification strategy that establishes objectives, stages the complexity, and identifies coverage points.

Table 1. Possible proof status of formal verification

Proof status	Description
Unprocessed	The property is excluded from the proof target, even if declared
Undetermined	Neither full pass nor counterexample found over the run time
Counterexample	Property violated in at least one legal case
Proven	Property exhaustively proven in all the legal cases
Cover	The intended behavior can occur at least once
Unreachable	The intended behavior is never possible

4 Reconstruction of FSM-Like Datapath Behavior

In Digital Signal Processing (DSP) applications, numerous digital circuits exhibit behavior similar to FSM, where the next state is determined by the current state. This characteristic is straightforward to replicate in RTL designs due to the presence of memory elements such as registers that can hold state information. However, in C/C++ models, which are inherently untimed, managing state transitions to drive the next state can be challenging. In this section, the FEV process on checking the functional equivalence of a Multiplier-Accumulator (MAC) is described in detail.

The MAC unit finds extensive application across various DSP fields, including but not limited to audio and speech processing, image and video compression, telecommunications, radar and sonar systems, as well as biomedical signal processing. Specifically, it enables rapid computation for tasks such as filtering, convolution, and Fourier transforms in embedded systems. The main purpose of MAC is to repetitively add the product between two input signals to previously obtained intermediate results of the same nature. More precisely, such a functionality can be modeled under a mathematical point of view, according to Eq. 2. Given $a(k)$, $b(k)$ as input signals sampled at k-th cycle, the multiplication operation produces an intermediate result which is added to the sum of those computed during the $k-1$ previous cycles.

$$result = \sum_{k=0}^{N} a(k) \cdot b(k) \tag{2}$$

4.1 Specifications

The Device Under Verification is a Floating-Point MAC compliant to the IEEE-754 Standard for Floating-Point Arithmetic [7], patented by STMicroelectronics [19], whose interface and computational block diagrams are shown in Fig. 4. The block accepts three floating-point operands (namely *fp_a*, *fp_b*, *fp_c*) and a starting condition triggering a new operation to execute, that is indicated by *fp_opcode_i* (refer to List. 1.1 and List. 1.2 in [15] for C and RTL pseudo-codes).

Operational stability requires that, once the execution phase begins, the input signal *fp_op_start* remains inactive, while the input operands and the opcode signal retain their values until the completion of the computational phase. The block supports both straight and recursive arithmetical operations, involving multiplication, addition, and subtraction. On the output side, two distinct floating-point results (*fp_m* and *fp_z*) are provided, respectively containing the sampled outcomes produced by the multiplier and the adder/subtractor blocks.

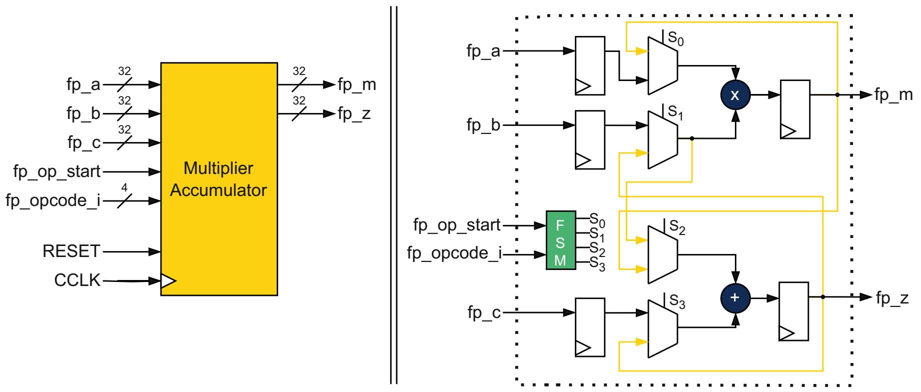


Fig. 4. Interface and computational block diagrams of the MAC

4.2 Verification Strategy

The verification aimed to affirm the RTL implementation's equivalence with its C model. The circuit comprised a pipelined floating-point multiplier and a combinatorial floating-point adder from a third-party IP, complicating direct formal verification. Focus thus shifted to verifying the control logic and feedback mechanism, using fixed-point behavioral models to represent the floating-point units analytically. Recursive operations in the RTL allow reusing previous outputs as operands for current computations, challenging to replicate in untimed C/C++ models due to their instantaneous computation on the same formal analysis cycle. While appropriate latency was easily introduced in model comparisons to properly handle the gap in pipelined designs, the formal tool lacked features to enable the C/C++ model to retain past values.

The most promising strategy consisted in extending the interface of the C/C++ model (see List. 1.1 in [15]), in a way that output values could be brought back as operands for recursive operations. Within this framework, SVA assumptions were essential to ensure the feedback continuity, forcing that *fp_m_in* corresponds to the prior cycle's *fp_m_out*, and similarly for *fp_z_in* and *fp_z_out*, as delineated in List. 1.3 in [15].

Further strategies to achieve complete convergence in useful time consisted in scaling down the bit-width of the arithmetical operands, from 32-bit to 8-bit and 16-bit. In fact, design scaling simplifies the state space and thus accelerates convergence in formal verification by diminishing the computational complexity and the number of potential states to explore. Moreover, a case-splitting strategy, guided by opcode values, was implemented to isolate and address the most challenging cases for the verification tool, uncovering bottlenecks tied to specific algorithmic attributes. Consequently, manual assertions were crafted to refine the scope of auto-generated properties, as reported in List. 1.4 in [15].

To alleviate the verification load and ensure comprehensive equivalence, assertions were reformulated. This entailed incorporating the triggering condition and confirming the initial congruence of output signals from the preceding cycle. Stability of the summation outputs was verified over the calculation span of six cycles, whereas the multiplication output from the RTL was validated to not changing within the five-cycle interval post-triggering, concurrently maintaining equivalence with the C model in the subsequent cycle (see List 1.5 in [15]).

4.3 Results

Table 2 encapsulates the verification methodology for the MAC block. Despite some instances of *undetermined* proof results arose, we ultimately established complete equivalence across all cases by advancing through the verification sequence, confirming the functional correctness of the control part of the MAC block. Table 3 reports the run-times obtained at the last verification stage by bit-width value, illustrating the substantial influence of design scaling on formal tool performance. Additionally, Table 4 provides run-times from an earlier verification phase, demonstrating how case-splitting helps identify the most challenging scenarios for proof, namely recursive operations.

Table 2. Staging complexity in the verification of the MAC

Stage	Description
1	Verification of the control logic with fixed-point behavioral operators
2	Reconstruction of the feedback mechanism using SVA assumptions
3	Scaling down the bit-width to reduce the complexity of the formal problem
4	Application of case-splitting technique and manually written properties
5	Reformulating the manually written assertions to lighten the verification load

Table 3. Final run-times of the MAC verification considering all the opcodes

Opcode	8-bit	16-bit	32-bit
$[0x00,\ 0x0C]$	17.27 s	58.66 s	99.32 s

Table 4. Run-times at stage 4 of the MAC verification sequence (* stands for *undetermined* proof result, while apex symbol indicates the value at the previous cycle)

Opcode	Equation	8-bit	16-bit	32-bit
$0x00$	$fp\_z = fp\_b + fp\_c$	0.32 s	0.50 s	0.59 s
$0x01$	$fp\_z = fp\_b - fp\_c$	0.35 s	0.64 s	1.42 s
$0x02$	$fp\_z = fp\_z' + fp\_c$	0.34 s	1.44 s	1.34 s
$0x03$	$fp\_z = fp\_z' - fp\_c$	0.33 s	1.39 s	1.88 s
$0x04$	$fp\_z = fp\_m' + fp\_c$	$\approx$ 12 h *	$\approx$ 12 h *	$\approx$ 12 h *
$0x05$	$fp\_z = fp\_m' - fp\_c$	2.85 s	4.69 s	17.81 s
$0x06$	$fp\_m = fp\_a \cdot fp\_b$	7.54 s	39.57 s	131.97 s
$0x07$	$fp\_m = fp\_m' \cdot fp\_b$	$\approx$ 12 h *	$\approx$ 12 h *	$\approx$ 12 h *
$0x08$	$fp\_m = fp\_z' \cdot fp\_a$	2.62 s	21.35 s	0.88 s
$0x09$	$fp\_z = fp\_a \cdot fp\_b + fp\_c$	10.62 s	91.52 s	175.57 s
$0x0A$	$fp\_z = fp\_a \cdot fp\_b - fp\_c$	10.04 s	76.93 s	186.11 s
$0x0B$	$fp\_z = fp\_z' + fp\_a \cdot fp\_b$	13.4 s	63.97 s	530.02 s
$0x0C$	$fp\_z = fp\_z' \cdot fp\_a + fp\_c$	11.56 s	65.89 s	211.05 s

5 Decomposition of a Complex Cone of Influence

Utilizing the Jasper^{TM} C2RTL App facilitates equivalence checking to ascertain the functional correctness of an RTL design without necessitating manual property specification. Nevertheless, the automatic generation of end-to-end properties aimed at confirming output signal consistency under equivalent inputs can engender an intricate Cone Of Influence (COI). The COI, pivotal in formal verification, circumscribes the relevant RTL logic impacting a given property, enabling the exclusion of non-influential logic (refer to Fig. 5).

A case study on an STMicroelectronics-designed pipelined, frequency-tunable, and programmable-gain tone generator was undertaken to investigate methods for decomposing the COI in the context of challenging automatically generated properties. The tone generator is crucial for calibrating audio DSP systems, developing signal processing algorithms, and testing telecommunications networks.

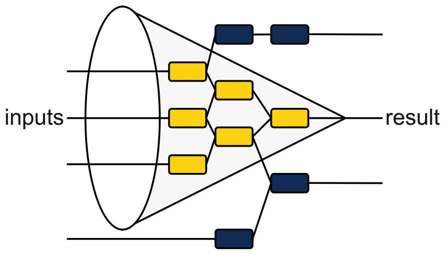


Fig. 5. Conceptual representation of the Cone Of Influence

5.1 Specifications

The tone generator accepts inputs such as the tone setup choice *mode_i* (either *single* or *double* tone), phase steps *$\Delta\Phi$_single* and *$\Delta\Phi$_double*, and programmable gains *gain_single* and *gain_double*, as shown in Fig. 6. It then generates outputs that consist of either one or two tones in the in-phase (I) and quadrature (Q) components, *Y_I* and *Y_Q* respectively. The phase step sets the incremental change in phase between successive samples, thereby setting the frequency of the tone(s), while gain controls the amplitude scaling applied to the output signal (see C and RTL pseudo-codes in List. 1.6 and List. 1.7 in [15]).

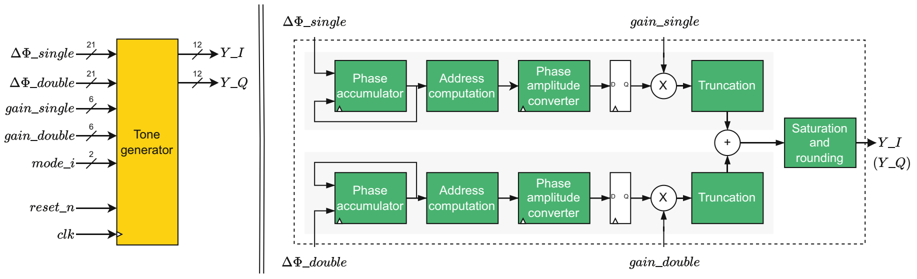


Fig. 6. Interface and computational block diagrams of the tone generator

The protocol governing input signals mandates static values within their defined legal ranges throughout execution:

- *mode_i* signal assumes values within the set $\{0,\ 1,\ 2\}$, where 0 denotes the idle state, 1 corresponds to single tone mode, and 2 to double tone mode.
- *gain_single* and *gain_double* signals are constrained to $[0, 63]$ and $[0, 31]$.
- *$\Delta\Phi$_single* and *$\Delta\Phi$_double* signals are restricted to the range $[0, (2^{21} - 1)]$.

5.2 Verification Strategy

The verification's primary objective was to establish the complete equivalence of the RTL implementation of the tone generator with its corresponding C/C++ model. Central to the block's functionality is a phase accumulator unit designed to iteratively compute the subsequent phase value, ($\Phi(t+1)$), from the current phase, ($\Phi(t)$), and the input phase increment, ($\Delta\Phi$), as described by the equation ($\Phi(t+1) = \Phi(t) + \Delta\Phi$). To manage the inherent feedback within the phase accumulator, a verification strategy analogous to that delineated in Sect. 4 was employed. This strategy proved ineffective, except for large phase step values, primarily due to the extensive bit-width and the vast array of potential cases which could not be exhaustively verified.

Given the design's intrinsic architecture, reducing parallelism was not feasible without altering the models, a course of action avoided due to the potential for introducing errors. Consequently, an alternative strategy was adopted, which involved overconstraining the current phase value node in both the RTL and C/C++ models to accept any value within its legal range, independent of the phase step. The overconstraint ensured that the subsequent phase value would correspond to the overconstrained phase node's value from the previous cycle, thereby emulating the phase accumulator's functionality (see List. 1.8 in [15]). Overconstraining the internal phase node had not compromised the functionality of the circuit, since it affected both downstream and upstream logic.

Despite promising, overconstraining the internal phase node did not result as a completely satisfactory strategy because of the long time required to achieve the full proof. Because this was mainly due to the high complexity of the computational load, a more powerful technique, consisting in inserting extra assertions by leveraging intermediate equivalent points between the C/C++ and RTL models, was employed. While it may appear that adding more assertions could increase the workload for the verification tool, proven assertions at the intermediate key points can actually aid the formal tool in verifying more complex automatically generated end-to-end properties (see List. 1.9 [15]). In this case, intermediate equivalent points were placed at data processing stages (refer to Fig. 6), such as:

– Sample values coming out from the phase amplitude converter.
– Sample values scaled by the input gain and then truncated.

A more sophisticated and efficient verification strategy involved explicitly instructing the formal verification tool to utilize proven assertions at intermediate equivalent key points within the design. By doing so, these assertions act as simple blocks within a more complex chain of end-to-end properties. As the verification tool progresses through the smaller properties, it utilizes the *proven assertions* as *helper assumptions* for subsequent assertions in the verification chain. An end-to-end property is considered *proven* if all its helper assumptions are also *proven*. Consequently, the *assume-guarantee* method was employed as the terminal verification technique to expedite the attainment of a comprehensive proof. This approach mitigated the complexity of the global Cone Of Influence by partitioning challenging monolithic assertions into discrete, tractable formal verification sub-problems, each with a correspondingly narrowed COI.

5.3 Results

The application of FEV techniques successfully confirmed the functional equivalence between the C/C++ model and the RTL implementations. For the sake of clarity, proof convergence was achieved by following the verification sequence reported in Table 5. Table 6 reports the infeasibility of feedback reconstruction using SVA assumptions, highlighting the formal tool's difficulty with diminishing phase step values. Table 7 summarizes the run-times by verification stage and working mode, proving the advantage of determining the equivalence at intermediate points to aid the formal tool in achieving convergence.

Table 5. Staging complexity in the verification of the tone generator

Stage	Description
1	Reconstructing the feedback of the phase accumulator
2	Overconstraining the current phase node using SVA assumptions
3	Proving the equivalence at intermediate points inserting extra assertions
4	Applying the assume-guarantee to leverage equivalence at intermediate points

Table 6. Run-times at stage 1 of the tone generator verification sequence

Input phase step	Proof status	Run-time
2^{20}	Proven	17.6 s
2^{19}	Proven	62.8 s
2^{18}	Proven	29 min
2^{17}	Proven	≈ 12 h
2^{16}	Undetermined	≈ 24 h

Table 7. Run-times at different stages of the tone generator verification sequence

Single tone mode	Stage 2	Stage 3	Stage 4
Run-times	221 min	132 min	26 min
Proof status	Proven	Proven	Proven
Time reduction		40%	80%
Double tone mode	Stage 2	Stage 3	Stage 4
Run-times	48 h	111 min	33 min
Proof status	Undetermined	Proven	Proven
Time reduction		N/A	70%

6 Proving the Equivalence with a MATLAB®-derived C Code

Significant algorithmic differences between high-level C/C++ models and RTL designs present notable challenges in proving functional equivalence using FEV techniques. This is especially the case for C code derived from MATLAB® where complexity can increase due to several factors, such as the variations in data types and bit-width choices. Despite casting procedures are supported by the formal tool, it is highly recommended to minimize type discrepancies between RTL and C/C++ representations. This approach was employed in the verification of an Automatic Gain Control (AGC) design.

6.1 Specifications

The main purpose of an AGC circuit, within a receiver in a communication system, is to maintain a constant output amplitude level of a signal despite variations in the amplitude of input signal, as represented in Fig. 7. The device under analysis is an AGC (targeting IEEE 802.15.4g protocol [6]) designed by STMicroelectronics, governed by an FSM with datapath mechanism. Due to confidentiality constraints, detailed information about the specific block cannot be disclosed in this publication. The AGC accepts inputs from a Received Signal Strength Indicator (RSSI) block through *rssi_result* signal, providing an output gain value to a Programmable Gain Amplifier (PGA) using *gain* signal, as shown in Fig. 7. Configuration of the AGC is achieved by setting the *mode* signal, initializing the gain with *start_gain*, and adjusting the gain using *gain_step*.

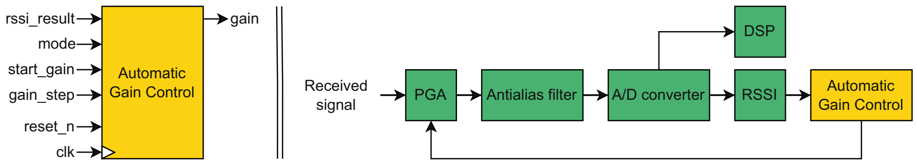


Fig. 7. System level representation of the Automatic Gain Control

6.2 Verification Strategy

Significant algorithmic differences between the two models precluded to leverage intermediate equivalent key points to aid the formal tool. Consequently, beyond feedback reconstruction technique described in Sect. 4 to emulate the FSM behavior, further modifications were implemented. To alleviate verification overhead, all *double* data types in the C/C++ model - automatically converted from MATLAB® codes using MATLAB Coder [12] - were converted to *int* data types to align with the RTL design specifications. This conversion necessitated the creation of new functions within the C/C++ code, which are listed in Tab. 10 in [15]. Additionally, to prevent state explosion issues commonly associated with counters, their maximum count values were deliberately constrained to zero to avoid multiple accumulation of samples. This cap was not overly restrictive, as the accumulation underwent separate verification from the gain computation under specific input scenarios, namely the main target of the verification process.

6.3 Results

The verification of this case was particularly challenging due to algorithmic divergences between the high-level and low-level models and the design's complex control logic. Table 8 outlines the verification sequence utilized for the AGC block, which ultimately resulted in the run-times presented in Table 9. Despite the full equivalence was not proven in all cases, the application of FEV techniques resulted valuable by quickly identifying two mismatches between the C/C++ model and the RTL design, corresponding to subtle overflow cases that were not discovered during previous UVM dynamic simulations.

Table 8. Staging complexity in the verification of the AGC

Stage	Description
1	Reconstructing the feedback mechanism to emulate the FSM behavior
2	Converting the data types of the C/C++ model from *double* to *int*
3	Disabling counters to avoid multiple accumulation of samples

Table 9. Run-times at the final stage of the AGC verification sequence

Mode	Description	Proof result	Run-time
Mode_1	Fixed-gain configuration	Proven	0.34 s
Mode_2	Gain can only decrease	Proven	121.74 s
Mode_3	Gain can only decrease until a certain value	Proven	754.10 s
Mode_4	Gain can both increase and decrease	Undetermined	≈ 48 h
Mode_5	Gain is determined by non-trivial RSSI conditions	Undetermined	≈ 48 h

7 Conclusion

This paper presented a case study on applying various FEV techniques within the context of verifying the functional equivalence between three-real world high-level C/C++ models and RTL designs using the Jasper^{TM} C2RTL App. This innovative approach enables exhaustive exploration of the design state space, potentially revealing bugs that traditional verification methods might miss. Moreover, by utilizing high-performance formal engines, the verification time for typical DSP components has been significantly reduced - from months to just a few weeks per case study -compared to UVM dynamic simulations, specifically:

– UVM environment setup traditionally requires six weeks, whereas C2RTL preparation, including port mapping, adaptation of C/C++ models, and verification plan formulation, is completed within one week.
– Test development in UVM extends over five weeks, in contrast to the two weeks needed for incorporating appropriate constraints in C2RTL. This entails specifying legal input signal values and protocols, methodically exploring the design state space, and applying effective verification techniques to ensure convergence.

– Debugging in UVM, which involves analyzing dynamic simulation waveforms, typically spans two weeks. Conversely, C2RTL reduces this to a matter of days, benefiting from the provision of succinct counterexample waveforms and facilitated root cause analysis.

Customizing the formal tool to accommodate the specific characteristics of each DUV proved to be a non-trivial task. There is no replacement for the verification user's knowledge of the expected behavior and the selection of appropriate techniques to assist the tool in handling FSM-like behaviors, large COI and significant algorithmic difference between high-level C/C++ models and the RTL designs. In conclusion, this paper has demonstrated that the strategic application of FEV techniques, facilitated by the Jasper^{TM} C2RTL App, significantly enhances the efficiency and effectiveness of DSP component verification.

References

1. Albin, K.: Oracle labs: the cost of SoC bugs. In: Design and Verification Conference and Exhibition, U.S. (2016)
2. Bergeron, J.: Writing Testbenches using SystemVerilog . 1st edn. Springer, (2006)
3. Cadence Design System: Jasper C to RTL Equivalence Checking App User Guide (2023)
4. Cadence Design System: Jasper Engine Selection Guide (2023)
5. Formality Equivalence Checking. https://www.synopsys.com/glossary/what-is-equivalence-checking.html. Accessed 15 June 2024
6. IEEE: IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 3: Physical Layer (PHY) Specifications for Low-Data-Rate, Wireless, Smart Metering Utility Networks, pp. 1–252, IEEE Std 802.15.4g-2012 (2012)
7. IEEE-754: Standard for Floating-Point Arithmetic. IEEE Std 754-2008, pp. 1–58. IEEE (2008)
8. IEEE Computer Society and IEEE Standards Association Corporate Advisory Group: IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800^{TM}-2017). IEEE, New York (2017)
9. Jasper C Apps. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-c-formal-verification.html. Accessed 13 June 2024
10. Jasper C Apps. https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/jasperc2rtl. Accessed 14 June 2024
11. Jasper SEC App. https://www.cadence.com/zh_TW/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/jaspergold-sequential-equivalence-checking-app.html. Accessed 15 June 2024
12. MATLAB Coder. https://it.mathworks.com/products/matlab-coder.html. Accessed 19 Jun 2024
13. Mittal, V., Roy, S., Singhal A.: Embracing datapath verification with Jasper C2RTL App. In: Design and Verification Conference, India (2022)
14. Perry, D.L., Foster, H.: Applied Formal Verification: For Digital Circuit Design. 1st ed. McGraw Hill LLC (2005)

15. Raia, G., Vincenzoni, D., Rigano, G., Martina, M.: A Case Study on Formal Equivalence Verification between a C/C++ Model and its RTL Design: A Long Companion Version. Zenodo (2024). https://doi.org/10.5281/zenodo.12591803
16. Seligman, E., Schubert, T., Kirankumar, M.: Formal Verification: An Essential Toolkit for Modern VLSI Design, 1st edn. Morgan Kaufmann Publishers Inc, San Francisco (2015)
17. The 2022 Wilson Research Group Functional Verification Study (Part 8). https://blogs.sw.siemens.com/verificationhorizons/2022/12/12/part-8-the-2022-wilson-research-group-functional-verification-study/. Accessed 13 June 2024
18. The 2022 Wilson Research Group Functional Verification Study (Part 12). https://blogs.sw.siemens.com/verificationhorizons/2023/01/09/part-12-the-2020-wilson-research-group-functional-verification-study-2/. Accessed 13 June 2024
19. Vincenzoni, D., Raffaelli, S.: Circuit for performing a multiply-and-accumulate operation. (10089078, 3299952,10437558) (2016)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Tutorial Papers

A Pyramid Of (Formal) Software Verification

Martin Brain[1](✉) and Elizabeth Polgreen[2]

[1] City, University of London, London, England
martin.brain@city.ac.uk
[2] University of Edinburgh, Edinburgh, Scotland
elizabeth.polgreen@ed.ac.uk

Abstract. Over the past few years there has been significant progress in the various fields of software verification resulting in many useful tools and successful deployments, both academic and commercial. However much of the work describing these tools and ideas is written by and for the research community. The scale, diversity and focus of the literature can act as a barrier, separating industrial users and the wider academic community from the tools that could make their work more efficient, more certain and more productive. This tutorial gives a simple classification of verification techniques in terms of a pyramid and uses it to describe the six main schools of verification technologies. We have found this approach valuable for building collaborations with industry as it allows us to explain the intrinsic strengths and weaknesses of techniques and pick the right tool for any given industrial application. The model also highlights some of the cultural differences and unspoken assumptions of different areas of verification and illuminates future directions.

1 Introduction

Software verification is a large and diverse area of computer science research. Topics covered range from low-level, practical issues such as understanding the exact behaviour of various hardware and software constructs through to high-level, theoretical issues of expressibility and the limits of what is computable. The diversity of and connections between these areas can make it hard to understand and appreciate the full power and applicability of the ideas. This is compounded by the existence of several different academic traditions or schools each of which have their own terminology and foundations.

The field also has a strong culture of tool development, leading to a range of powerful academic and commercial tools. However potential users (both academic and commercial) are often faced with the problem of understanding how these various tools relate to each other and their various strengths and weaknesses. Their problem is not which tool to pick but *on what basis to make their decision*. For experienced academics and researchers, the answers are often ‘obvious’ but, again, this requires a broad and comprehensive knowledge of the different approaches and traditions of verification.

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 393–419, 2025.
https://doi.org/10.1007/978-3-031-71177-0_24

This paper describes our approach to bridging this gap and communicating the 'big-picture' of software verification without requiring people to read many papers, attend numerous conferences or develop multiple tools. The approach has been developed and used in numerous industrial partnerships as well as in undergraduate and post-graduate teaching. It has provided a simple way of structuring the explanation of what we do and how this fits with particular organisations' needs. We regard this as a successful and efficient way of bridging the (common) divide between research and practice.

As with all overviews, there are exceptions and caveats to all of the classifications we give. There are also systems which combine multiple techniques, whose classification is debatable or ambiguous. If such exceptions and combined techniques do not already exist, they will likely do so soon as they represent novel research directions. In this regard, this paper should be thought of as a guidebook or a phrase book rather than an atlas or dictionary. Our goal is to describe the common 90% of papers in a field rather than the exceptional 10%.

We assume a base understanding of computing, but this paper is intended to be readable by commercial developers and does not assume prior knowledge of verification or theoretical computer science. We hope it will communicate:

- Our pyramid model, which gives a fundamental trade-off for software verification tools (Sect. 2).
- The kinds of tools available, their intrinsic strengths and weaknesses, and enough of the culture and terminology to communicate with and evaluate tools from the relevant research community (Sect. 3).
- The different ways of giving specifications (Sect. 4).
- How to select the right kind of tool(s) for a particular practical problem (Sect. 5).

2 A Pyramid of Verification

In the most general sense, verification is the process of checking the properties of a thing against a set of criteria. In our context, we will refer to the thing being checked as the *program* and the criteria as the *specification*. Figure 1 illustrates the verification process. The primary inputs are the specification and the program plus the verification process uses an amount of compute resource and human effort. The ideal outcome is that the system is verified but it is also possible that one or more defects are detected or that the verification is inconclusive and the final result is unknown. From a practical point of view, unknown is probably the worst outcome as the effort is expended without producing useful evidence. However, as we will see, reliably avoiding the unknown outcome turns out to be challenging.

Representing the program. Our primary focus is when the program (thing we are verifying) is a piece of software written in an imperative language. However many of these techniques have been successfully applied to hardware, parallel

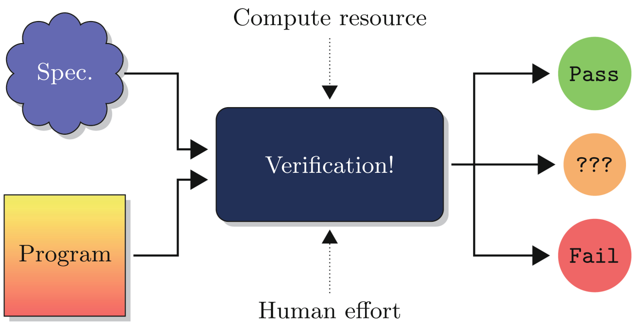


Fig. 1. The verification process in principle

and distributed software as well as more abstract models of computation such as protocols, process calculi, automata, cyber-physical systems, transition systems, etc. One common view in software verification, and that we will use in the following sections of the paper, is that a program is a *description* or *representation* of a set of traces. A *trace* is a single execution of the program; a sequence of states showing the step-by-step execution of the program. Running a program is computing a single trace. For a program of any size it is infeasible to compute all of the traces, so the set of all traces is a mathematical idea rather than something that is ever directly computed. This viewpoint is useful as it allows the program and the specification to be seen as the same kind of mathematical objects; sets of traces. It also gives an idea of *over-approximation* and *under-approximation* of a program. These are sets of traces that contain or are contained in the set of traces of the program.

Representing the specification. The specification is the set of criteria we check the program against. Software verification tools have traditionally focused on *universal* specifications; those of the form "every execution of the program must" or conversely "there must be no way of" rather than *existential* ones, "there must be at least one trace that". Other, more complicated, properties can be specified that reason about the interaction of traces (hyperproperties) or the likelihood of certain traces (probabilistic properties), but we do not discuss those here. If the program is understood as a representation of a set of traces then the specification can be understood as a representation of the set of all traces that have a required property. Formal verification of a universal specification can then be conceptually reduced to checking that the set of all program traces is *included* in the set of specified traces. We give further discussion on the ways of representing specifications in Sect. 4.

Given a program and a specification to verify it against, the ideal verification tool would be:

Automatic (run with no human interaction),
Never miss bugs (only say verified if the system meets the specification), and

Never give false alarms (always say verified if the system meets the specification).

Unfortunately, if the specification includes any notion of reachability ("if the program ever ...", "when the program ...") then Rice's Theorem [73], a consequence of Turing's famous result on the Halting Problem [78] means that it is not possible to create a verification tool that has all the desired properties[1] in all cases. As almost all significant specifications include some notion of reachability or location, this gives us the fundamental trade-off at the heart of software verification:

> It is not possible to create a verification tool which can take *any* program and *any* specification and *automatically* give an answer in a finite amount of time guaranteeing *no missed bugs* and *no false alarms*.

As with all applications of theoretical results to the real world, we should be mindful of the caveats. Turing's result applies to a theoretical model of computing which has an infinite state-space. We will assume that the state-space of programs is so large that is effectively infinite[2] and that this result applies. The subtleties of quantification are also important. This result applies to one, or a finite number of verification tools working on *all* programs and *all* specifications. For a particular program and specification, there is a verification tool that can automatically give a full answer (although writing it might require performing the verification by hand and then writing a program that will simply print the answer). Likewise, there are large sets of programs and specifications for which this is possible. For example, if the specification is "the program terminates", then any acyclic program (no loops or recursion) can easily be automatically verified in the time it takes to read the program and check it is acyclic.

Although theoretical computer science shows that it is not possible to build a universal verification tool, it is easy to get surprisingly close:

- 'Automatic' and 'no missed bugs' can be achieved by over-approximating the set of invalid traces (traces of the program that do not meet the specification). A tool that prints `verification failed` for all inputs would be the simplest example.

[1] We avoid the terms "sound" and "complete" as there is a cultural bias for authors to claim that their technique is "sound but not complete". This leads to two, opposite definitions of these terms. Some authors use "sound/complete (for proof)" while others use "sound/complete (for refutation)". No missed bugs is "sound for proof" and "complete for refutation" while no false alarms is "complete for proof" and "sound for refutation".

[2] Clearly, any individual computer has physical limits on the number of bits that can be stored and thus has a limit on the state-space of a program running on it. However, the state-space of a program running on a processor with 640KB of memory (up to $2^{5242880}$ states) may be regarded as *practically* infinite as it contains more states than particles in the universe.

- 'Automatic' and 'no false alarms' can be achieved by under-approximating the set of invalid traces. A tool that prints `verification succeeded` for all inputs would be the simplest example.
- Finally 'no missed bugs' and 'no false alarms' can be achieved by getting a human to create a formal proof of the verification and checking it.

As it is easy to achieve two of the three criteria, we can view verification tools as starting with up to two of these and then trading computational effort to achieve the third for an ever-increasing set of programs and specifications. For example, *over-approximate* tools (automatic and no missed bugs) use computation to reduce the number of false alarms. *Under-approximate* tools (automatic and no false alarms) use computation to reduce the number of missed bugs. *Human-assisted* tools (no missed bugs and no false alarms) use computation to reduce the amount of human effort required.

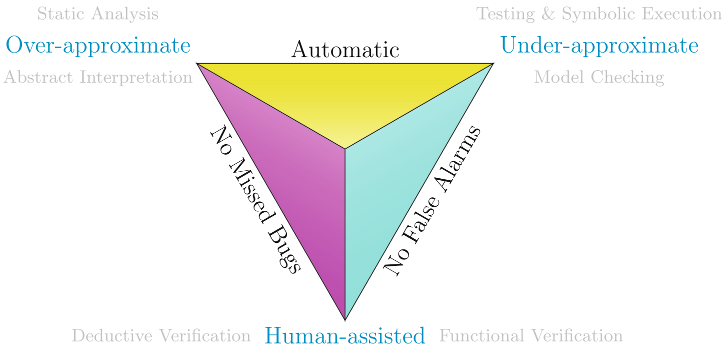


Fig. 2. The software verification pyramid with the six schools.

Figure 2 illustrates this trade-off with a three-sided pyramid (viewed from above). Each base edge of the pyramid represents one of the three attributes; automatic, no missed bugs and no false alarms. Corners on the base of the pyramid represent each of the three approaches; over-approximate, under-approximate and human-assisted. The top of the pyramid, in the centre of the diagram, represents the ideal system with all three attributes. Computation effort is then used to 'climb' the pyramid towards the top, with different techniques giving different routes up their chosen edge (or face).

There are many other dimensions on which software verification techniques can be classified, and many other trade-offs that are necessarily made by different tools. For instance, one could also consider how properties are specified, the categories of systems that are analysed, and the usability of the system and specification language. Our pyramid model is not intended to be exhaustive, but it is intended as a useful starting point when making the initial choices about how to solve a verification problem.

Table 1. Cultural attributes of the six schools.

	Over-Approximate		Under-Approximate		Human-Assisted	
	Static Analysis	Abstract Interpretation	Testing & Symbolic Execution	Model Checking	Deductive Verification	Functional Verification
Program	Procedural or O.O.	Procedural	Procedural or O.O.	Procedural or O.O.	Subsets of procedural	Functional
Commmon Means of Specification	Builtin	Annotation linked to the abstraction	Generally annotation	Annotation or external	Annotation	Type as annotation
Common Type of Specification	Data flow, aliasing, type, shape, taint	Value, shape, resource, data flow	Value, WCET, resource	Value, temporal, modal, liveness	Value, shape, termination, resource	Type, termination
Mathematical Foundations	Ad-hoc / operational semantics	Order theory	Ad-hoc / transition systems	Transition systems	Logic	Type theory
User Skill Required	Minimal	Low/Medium	Low	Medium	High	Very high
Compute Required	Minimal	Low/Medium upwards	Medium upwards	Medium/High upwards	Low/Medium	Low
Typical Output	Algorithm dependent	Alarms or abstract domains	Error traces	Error traces	Proof or local counter-examples	Type-checking errors
Major Systems	Lint[55], Coverity[1], Fortify[5], FindBugs[4], CPPCheck[2]	Astrée[33], Polyspace[7], Infer[57], Code Contracts[66]	CREST[3], JPF[50], Pex[77], KLEE[18]	CBMC[60], Blast[51], *SMV[22], CPAchecker[13]	SPARK[8], Dafny[64], Frama-C[34], Malpas[6], Esc/Java[45]	Coq[12], PVS [70], Agda[69], Isabelle/Hol[68]

3 Six Schools of Verification

Our pyramid model allows us to compare and contrast the different academic traditions in and aligned with software verification. Each of these represents a separate lineage of thought, approach and community, although there is interaction, overlap and cross-fertilisation between them. This section surveys them and attempts to give a qualitative assessment. Of the six schools, not all would regard themselves as software verification but all have an important impact on the field. As with all qualitative assessments, there is an element of subjectivity and no doubt researchers and practitioners from each of the fields have different views; the aim of this document is to provide an overview rather than a definitive assessment. There are also exceptions to most of the distinctions and classifications and our statements should be taken as normative or cultural observations rather than hard limits.

Table 1 gives the six schools with various cultural attributes. The kinds of programs verified, and the kind of specification (see Sect. 4) normally considered are given along with the mathematical formalisation used to express theoretical results and algorithms. It also gives quantitative valuation for the level of user skill required to apply the tools effectively and the computational requirements of the tools. Finally, it gives a few of the prominent or significant tools from each of the areas. Many tool suites include tools from several of the different schools. These are listed under the school for their main tool or general approach.

As well as the traditional approaches of the six schools, there are also a significant number of combined techniques and tools. The simplest of these are pipe-lines that run different tools in sequence (or sometimes in parallel), feeding the results of one into the next. More sophisticated, synergistic combinations also exist and will likely form a major part of the future of software verification.

Figure 3 gives an example program in a C-like language. The specification is given as three assertions (an example of what Sect. 4 describes as an annotation specification). They express:

1. All array accesses are in bounds, at line 6
2. `last` is in `a` at end at line 13
3. If `found` is non zero at end then `a[last] == target` at line 14.

Try working out which of these can be verified. If they can be verified, what knowledge about the program is required to show this? If not, how would you provide a counter-example? Answers can be found here[3].

```
 1 (int, int)
 2 count (Array a, int target)
 3 {
 4   int found=0, last=-1, i=0;
 5   while (i < a.length()) {
 6     assert(0≤i<a.length());
 7     if (a[i] == target) {
 8       found = found + 1;
 9       last = i;
10     }
11     i = i + 1;
12   }
13   assert(0≤last<a.length());
14   assert(found≠0 ⟹ a[last]=target);
15   return (found, last);
16 }
```

Fig. 3. The running example program. The specification we wish to check is in the form of 3 assertions.

3.1 Static Analysis

One of the oldest fields of program analysis and verification is static analysis. Unfortunately, this term is used in at least two ways. In the general sense, it refers to program analysis performed without running the program (hence static). In this sense, the majority of the techniques of software verification can be regarded as static analysis. In the narrow sense, it refers to a specific set of algorithms and techniques that are explicitly referred to as static analysis (as opposed to any other name), and are often developed with and to serve the needs of compilers, both in improving their warnings (verification against a specification) and optimising the code they generate.

[3] Assertion 1 always holds (i.e., is always true) as the initial assignment to `i` is `0`, it is only ever incremented and the assertion appears immediately after the loop condition checks that `i < a.length()`. Assertion 2 can fail, as if the array does not contain the `target`, `last` will still have its initial `-1` value. Assertion 3 always holds as the only place `last` is assigned guarantees this property, although some non-trivial reasoning about reachability is required.

The traditional focus of this area has been on over-approximate techniques. Spurious warnings are fine if not too numerous, likewise missing an optimisation is much more acceptable than miscompiled code based on a flawed assumption. The need to keep compilation fast and robust has meant the field has focused on fast and robust techniques for common problems such as finding uninitialised variables, eliminating possible aliasing between pointers and data flow for scheduling and optimisation rather than more complex specifications.

The first static analysis tools began to emerge in the 1970 s, beginning with Lint [55], developed at Bell Labs in 1978, which flagged suspicious constructs in code that could be suggestive of a bug in the code. There have been many versions of lint developed for many C and C++ compilers, and "linter" is sometimes used as a generic term for static analysis tools based on this paradigm of flagging patterns indicative of programming errors, bugs, stylistic errors and suspicious constructs. For example, Lint would flag this if statement as a suspicious construct since it always evaluates to true, which is probably not what the programmer intended:

```
unsigned x;
if(x < 0) ...
```

These static analysis tools can be broadly described as lexical scanners that look for patterns in code that are likely to be defects or vulnerabilities. Static analysis tools do not treat the process of finding bugs in software as a logical problem; none of the analyses performed by these static-analysis tools involves constructing proof objects, and, as a result, these static analysis tools are not able to discover many of the complex bugs that can be discovered by other verification tools. That said, there is a broad range of types of bugs that can be discovered through this kind of code analysis, many of which represent serious vulnerabilities, for example, potential buffer overflows.

To mitigate the large numbers of false alarms typically produced by most static analysis algorithms, modern static analysis tools may categorise the bugs into ranks by seriousness [4] (i.e., the likelihood of the bug being a serious vulnerability), or by applying filtering algorithms to the results [24,76].

Industry: Static analysis tools like the commercial Coverity [1] and open-source CPPCheck [2] are commonly used in industry and have been used to find bugs in projects like Mozilla [35].

Running Example: Static Analysis cannot check the specifications given in the running example. It could, however, flag simpler properties. For instance, in a typed program, it could flag if the comparison `i < a.length()` compares an integer with an unsigned integer.

3.2 Abstract Interpretation

The field of abstract interpretation started with a series of papers [30–32] on the theory underpinning a range of static analyses. These proposed using the tools of order theory (partially ordered sets, lattices and Galois connections) to separate the analysis into two components: domains which track the information required for the specific analysis and abstract algorithms describing how the analysis is performed.

The *domain* describes a data-structure that is used to represent an over-approximate summary of the state of the program at a given point. Traditionally there will be an instance of the domain for every program location. For example, the *constant domain* contains a map from variables to constant values (plus flags for "not a constant" and "no value assigned"). If the map has `found` $\rightarrow$ `0` then we know that every time that location is reached, `found` will be `0`. The *interval domain* stores a map from variables to intervals (plus a "no value assigned" flag). If `i` $\rightarrow [0, 10]$, then we have a bound for the possible values for `i`. This can represent all the cases that a constant domain can and also represent things it can't so we say it is a more precise representation (i.e. less of an over-approximation). Domains have been created for a wide range of different analyses; data flow analysis, constant propagation, pointer analysis, etc. [21,74,81]. Creating a new kind of analysis can be as simple as specifying an appropriate domain.

The choice of domain depends on several factors. Using a more detailed domain (i.e. less of an overapproximation) can reduce the number of false alarms but requires additional computation. Often it is necessary to choose a domain that can precisely express the specification and the reasons why it is true. For example, if the specification includes proving bounds on variables then an interval domain is a good choice. However, if the specification includes proving the equality of variables, then intervals are unlikely to be very useful as they cannot represent the relationship between variables. The mathematics of abstract interpretation allows domains to be combined in various ways. Selecting the right combination of domains for a particular program and specification is one of the more advanced skills that can boost the effectiveness of abstract interpretation.

The second part of abstract interpretation is the analysis algorithms. These are stated in terms of mathematical operations on the domain, typically:

JoinCombines two instances of the domain to create a new instance that over-approximates both. For example, if one instance has `x` $\rightarrow [1, 4]$ and the other has `x` $\rightarrow [6, 8]$ then in the join `x` $\rightarrow [1, 8]$ – the smallest interval that contains both inputs.

TransformTakes one instance and an instruction and creates a new instance which over-approximates the effects of the instruction. For example if an instance contains `x` $\rightarrow [1, 8]$ and `y` $\rightarrow [-4, 4]$ then the transformer for `z = x + y` would create a new domain containing `z` $\rightarrow [-3, 12]$ and all other variables mapped to the same as in the original instance.

WidenCreates an instance of the domain that over-approximates the fix-point of a series of instances. For example given a loop `for (i = 0; i < n; ++i)` the widen operator might create an instance with `i` $\rightarrow [0, MAX]$.

Using abstract operations such as these allows the analysis algorithms to be specified and implemented independently of the domain, giving another 'orthogonal' space of possibilities. Analysis algorithms are often characterised by *sensitivity* to various program constructs:

Flow Sensitive:the order of instructions in the program is followed.
Path Sensitive:the branch conditions are applied to improve precision.
Context Sensitive:the calling context of functions is considered.

By using a more sensitive algorithm, the amount of over-approximation can be reduced and the set of specifications that can be automatically verified, with no false alarms, is increased. Of course, this can only be achieved by increasing the amount of computation required, so the more sensitive the algorithm the more expensive it will be to use. One of the practical strengths of abstract interpretation is modularity; assuming that variables or parts of a program are independent of each other is an over-approximation. This fits naturally within abstract interpretation. It is also the basis of many techniques for improving scalability.

Industry: Abstract interpretation has been used in industry to prove the absence of bugs in flight control software [56], and to analyse worst-case execution time for microprocessors [42]

Table 2. Table showing the result of abstract interpretation on the running example, using an interval domain.

	Line	found	last	i	Assertion result
L4	found = 0				
L4	last = -1	[0,0]			
L4	i = 0	[0,0]	[-1,-1]		
L5	i < a.length()	[0,x]	[-1, x-1]	[0,x]	
L6	$0 \leq$ i < a.length()	[0,x]	[-1, x-1]	[0,x]	pass
L7	a[i]=target	[0,x]	[-1, x-1]	[0,x]	
L8	found=found+1	[0,x]	[-1, x-1]	[0,x]	
L9	last=i	[1, x+1]	[-1, x-1]	[0,x]	
L11	i=i+1	[0, x+1]	[-1, x]	[0,x]	
L13	$0 \leq$ last $\leq$ a.length()	[0,x]	[-1, x-1]	[0,x]	unknown
L14	found$\neq$0 $\implies$ a[last]=target	[0,x]	[-1, x-1]	[0,x]	unknown

Running Example: Table 2 shows the result of abstract interpretation being applied to the running example, using the interval domain. Abstract interpretation can prove assertion 1 is always true, but cannot prove that assertion 2 fails or that assertion 3 is always true.

3.3 Testing and Symbolic Execution

Testing compiling and executing code on some concrete input(s), has a dual role in development. It is both a verification tool (does the test give a result allowed by the specification) and a validation tool (does the test do what I expected). In its verification role, testing is most suitable for existential specifications (things of the form "The software must be able to ...") because it is an underapproximate technique, only exploring a subset of possible traces of the program. Dijkstra famously described this situation [37] by saying: "Program testing can be used to show the presence of bugs, but never to show their absence!" . Showing the absence of bugs is the same as saying that every trace of the program does not trigger any bugs; a "universal" specification (Sect. 2).

One pragmatic option is to try to explore a 'sufficient' set of traces so that if there are bugs there is a high probability they will be found. This is the motivation behind coverage metrics which have a notion of a set of traces that is 'sufficient' and likely include at least one example of each kind of program behaviour. Coverage metrics are widely used and give a base level of certainty, even for universal specifications. Testing and coverage metrics are a large topic and [49,71] give a summary of the current-state-of-the-art. Test inputs can be defined manually by developers or automatically generated using *fuzz testing* to try to increase these coverage metrics.

Symbolic Execution [58] is a verification technique that was developed in the context of testing. It aims to build a logical expression that describes all of the traces that have taken the same path through the control flow graph. Rather than test a single trace it covers a subset of traces whose behaviour is similar. The subset of traces is then tested against the specification by testing the satisfiability of the logical expression representing the subset of traces and the negation of the specification, using a satisfiability solver.

A symbolic execution tool keeps a set of symbolic states. Each of these contains its current location in the program, a map from variables to expressions (describing the space of value it could take) and a *path condition* which is a set of expressions giving the conditions that must hold for that path to be taken. The set initially contains a single symbolic state, at the start of the program, with every program variable mapped to set to a fresh logical variable and an empty path condition. The analysis proceeds as follows:

Assign: Assign symbolic variables to each variable in the program state. Evaluate program statements using the symbolic variables until you reach a branching condition

Branch: On reaching a branching condition, e.g., an if statement, choose a fork to explore. This places a constraint over the symbolic variables, e.g., in the case of an if statement, if we explore the path when the if condition evaluates to true.

Check: When the path reaches an assertion, pass the constraints over the symbolic variables to the solver along with a constraint representing the violation of the assertion.

Treating the memory as fully symbolic does not scale in practice, so symbolic execution engines typically implement a partial memory model in which writes are concretized, but reads are modelled as reads from symbolic memory, up to a certain finite size of memory, and beyond that are concretized [29].

A limitation of symbolic execution is the path explosion problem: the number of paths in a program typically grows exponentially with the program size (and in the case of unbounded loops may be infinite). This means that the larger the program, the less likely it is symbolic execution will manage to find a subset of paths that exercises a particular bug. There are several heuristics the community has developed to try to mitigate this problem, including merging paths [61], exploring paths in parallel [79], and use of different heuristics to control the order in which paths are explored. In addition, combined approaches exist: concolic testing is a successful combination of symbolic execution combined with testing (or *concrete* execution), which treats program variables as symbolic, but inputs as concrete. It is used in conjunction with constraint solvers to generate new concrete inputs with the aim of maximising code coverage. A good survey on symbolic execution techniques is [20]

Industry: Testing is ubiquitous in industry, and needs no specific citation. Symbolic execution is the underlying technique in many popular tools used in industry, for example, KLEE [18] has been used for a variety of applications including wireless sensor networks and exploit generation [19], JPF has been used at NASA on the Orion control software [72], and Microsoft's SAGE, using a combination of fuzzing and symbolic execution, is used to find bugs in Windows applications [48].

Table 3. Table showing the result of symbolic execution for one arbitrarily choosen path on the running example. α and t are symbolic variables, and the path constraints are constraints inferred from choosing a branch at each branching condition.

	path constraints	symbolic environment
L0 (assign)	true	$a \mapsto \alpha, target \mapsto t$
L4 (assign)	true	$\ldots, found \mapsto 0, last \mapsto -1, i \mapsto 0$
L5 (branch)	$0 < \alpha.length()$	$\ldots, found \mapsto 0, last \mapsto -1, i \mapsto 0$
L7 (branch)	$0 < \alpha.length() \wedge \alpha[0] \neq t$	$\ldots, found \mapsto 0, last \mapsto -1, i \mapsto 0$
L11 (assign)	$0 < \alpha.length() \wedge \alpha[0] \neq t$	$\ldots, found \mapsto 0, last \mapsto -1, i \mapsto 1$
L13 (check)	$0 < \alpha.length() \wedge \alpha[0] \neq t \wedge 0 \leq -1 < \alpha.length()$	$\ldots, found \mapsto 0, last \mapsto -1, i \mapsto 1$

Running Example: Table 3 shows a single path of symbolic execution being applied to the running example. We are able to prove that assertion 2 fails if the target is not in the array. No path in the graph is able to show assertion 1 or assertion 3 always hold.

Algorithm 1: Fixpoint

```
Result: Reachable states $R$
R = I;
while $1$ do
    if $R == R \wedge T$ then
        return R;
    else
        R = R$\wedge$T ;
    end
end
```

Algorithm 2: BMC

```
Result: Reachable states $R$
R = I;
i=0 ;
while $i < k$ do
    R = R$\wedge$T ;
    i++ ;
end
return R ;
```

3.4 Software Model Checking

Model checking involves constructing transition systems, and checking that these systems are *models* of given logical specifications. Originally, the field focused on specifications written in temporal logic [9], and systems that were manually specified using a process calculus giving a *labelled transition system* (LTS), an automata-like structure with states corresponding to states of programs and transitions to the possible developments of the system. Verification could be reduced to showing that the system's LTS was a model of the logic, giving rise to the name of the field.

Explicit State Model Checking explores the states of the LTS one at a time, using graph algorithms such as depth-first search, until either a counter-example for the property has been found or all reachable states have been explored. This is limited by the number of states so tends to be used on protocols, high-level designs and abstraction of software systems. SPIN [53], FDR [47] and the TLA [83] tools are example of this style.

Symbolic Model Checking uses boolean formulae to represent sets of states in the system, the transition relation and the properties we wish to check. For instance, a formula representing the initial states of the running example is $found = 0 \wedge last = -1 \wedge i = 0$. The most basic symbolic model checking algorithm for systems with finite-states computes a formula that represents the total set of reachable states (R) by starting with the initial state (I) and "unrolling" the transition relation (T) repeatedly until a fixed-point is reached, as shown in Algorithm 1. The formula can then be checked to see whether it satisfies the specification. Critical to the performance of these systems is the use of compact and efficient data structure to manipulate Boolean formulae. A form of decision trees known as Binary Decision Diagrams (BDDs) are a popular choice [17,23].

Algorithms which compute the fix-point are able to find all bugs provided computing the fix-point is possible (i.e., the system does not contain any unbounded loops) and the representation of the system as a transition relation is precise enough to capture any bugs (for instance, memory models are often approximate). If the system contains unbounded loops, the algorithm will never find a fix-point, and so approximations must be introduced in order to deal with these scenarios.

Bounded Model Checking (BMC) [14] is an under-approximate technique which, instead of computing the fix-point, unwinds a transition system to a finite bound k, and then checks for violations of the property within the states reachable in k steps. BMC thus only guarantees the absence of bugs that can be reached in k steps. For some systems, a completeness-threshold [25] can be computed such that it is guaranteed that, if no bugs exist within k steps, no bugs exist at all. However, computing completeness thresholds for unbounded loops amounts to solving the Halting problem and so the technique remains, in general, on the under-approximate corner of the pyramid. BMC in its original presentation begins by unwinding the transition system 1 step and looking for violations of the specification within 1 step, and then it unwinds the transition system 2 steps, and so on until it reaches k steps. However, many popular tool implementations will instead unroll the entire system minus any loops and recursion to their limits, and then unwind the loops to k steps, as shown in Fig. 4.

```
 1 (int, int)
 2 count (Array a, int target)
 3 {
 4   int found = 0, last = -1, i=0;
 5   /** First unwinding **/
 6   if (i < a.length()) {
 7     assert(0≤i<a.length());
 8     if (a[i] == target) {
 9       found = found + 1;
10       last = i; }
11     i = i + 1;
12
13     /** Second unwinding **/
14     if (i < a.length()) {
15       assert(0≤i<a.length());
16       if (a[i] == target) {
17         found = found + 1;
18         last = i; }
19       i = i + 1;
20
21     if(i < a.length())
22       /** Unwinding assertion **/
23       assert(0);
24     }
25   }
26   assert(0≤last<a.length());
27   assert(found≠0⇒a[last]=target);
28   return (found, last);
29 }
```

Fig. 4. BMC applied to the example program. The loop is unwound 2 times. The assertion at line 23 checks whether the loop is unrolled sufficiently for the input values

A significant development in BMC was the use of SAT-solvers [15] instead of BDDs. Once the formula representing reachable states has been constructed, a SAT solver can efficiently check whether a counterexample exists using this formula. This is similar to symbolic execution, but instead of formulae representing subsets of paths, we have one formula that represents all of the paths. Modern software model checking tools typically take as input either source code, or some intermediate compiled representation of the source code such as LLVM [63], and convert this into an LTS, and then use SAT-based model checking to check this LTS against the specification.

There are many model checking algorithms beyond those mentioned, such as IC3/PDR [16] and k-induction [39], as well as techniques for reducing the size of the state space, such as program slicing [82] and predicate abstraction [10], and many hybrid techniques that use combinations of symbolic and explicit representations for different elements of the program. We refer the reader to [40, 54] for a comprehensive survey.

Industry: Bounded model checkers for software, such as CBMC [60] have been applied to automotive software [75], verifying bootcode [28] and other industrial software at Amazon Web Services [27].

Running Example: BMC will be able to find a counterexample to assertion 2 with a bound of $k = 1$, which will show that if the array is size 1 and does not contain the target, `last` remains set to -1. It cannot prove assertion 1 and assertion 3 are true, although it can say that they hold up to a bound k.

3.5 Deductive Verification

Robert Floyd (working on flow-charts [46]) and Tony Hoare (working on programs [52]) developed equivalent approaches for manually constructing proofs of program correctness. Tony Hoare's presentation of the ideas as a logic was more widely adopted, leading the approach to be known as *Hoare logic*. Their approach contained two key ideas. First logical formulae are used to represent sets of states at a particular point in the program. The set contains all of the states that make the formula true. This gives the fundamental building block of Hoare logic; the triple:

$$\{Pre\}\ \texttt{Program}\ \{Post\}$$

where Pre and $Post$ are formulae and `Program` is a part of a program. The triple denotes the statement "If the state of the program meets the precondition (Pre is true) then after `Program` has been run the state will meet the postcondition ($Post$ is true)". Hoare logic gives a series of proof rules for how these triples can be constructed. One of these proof rules is the second key idea; that an inductive argument about an invariant set can be used to prove properties of loops:

$$\frac{\{Inv \wedge Cond\}\ \texttt{Body}\ \{Inv\}}{\{Inv\}\ \texttt{while (Cond) Body}\ \ \{Inv \wedge \neg Cond\}}$$

This rule formalises the argument : if the body of a loop takes a state in (the set described by) Inv to another state in Inv, and Inv is true before the loop, then it must be true after the loop. Inv is referred to as an *inductive invariant*. Inductive invariants are both the main strength of Hoare logic and its main cost. They allow a finite, small proof to reason about the behaviour of an unbounded number of traces. However, invariants often have to be created by humans as there is no way of creating suitable invariants automatically for all programs and specifications. This is why deductive verification is on the human assisted corner of the pyramid. Once the candidate invariants have been provided both they and the specification can be checked automatically. Tools such as Daikon [41] and Houdini [44] have had some success in suggesting routine invariants. The choice of loop invariant is dependent on both the program and the specification. If the invariant is too weak (describes a set with too many elements), it may not be sufficient to prove parts of the specification after the loop. If it is too strong (describes a set with too few elements) then it may not hold before the loop or may not be inductive. Devising a suitable loop invariant is a skilled task and is one of the reasons for the higher skill rating in Table 1.

Dijkstra [36] contributed various ideas to the field of deductive verification. He showed that some of Hoare's rules for constructing tuples could be replaced with an algorithm that transformed one formula into the other. The best known of these *predicate transformers* are the *strongest postcondition* which use the *Pre* condition and `Program` to compute the most precise *Post* and the *weakest precondition* which uses the *Post* condition and `Program` to compute the least restrictive *Pre* condition. Dijkstra also showed that these techniques could be used to build software from a specification so that it was provable correct and argued forcefully that these *formal methods*[4] were the only professional approach to software engineering. In doing so he provided not only the means but also the motivation for Hoare logic to be used as a verification technology rather than a purely theoretical construct.

Table 4. Verification conditions generated to check the verification conditions in Fig. 3, using the inductive invariant given in Sect. 3.5. For readability, we do not include the formulae that reason about variables which do not change.

$found = 0 \wedge last = -1 \wedge i = 0$	strongest post condition
$(found \neq 0 \implies a[last] = target)$	check invariant
$(found \neq 0 \implies a[last] = target)$	invariant
$i < a.length()$	loop body run
$a[i] = target \Rightarrow found' = found + 1 \wedge last' = i$	execute body
$i' = i + 1$	loop counter update
$0 \leq i < a.length()$	check assertion 1
$(found' \neq last' \implies a'[last'] = target')$	check invariant
$(found \neq 0 \implies a[last] = target)$	invariant
$\neg(i < a.length())$	loop exit
$0 \leq last < a.length()$	check assertion 2
$(found \neq 0 \implies a[last] = target)$	check assertion 3

[4] Software verification is a technique that is used by some formal methods. However there are formal methods which do not use it and uses of software verification in development methodologies which are not traditionally considered formal.

Early uses of Hoare logic were proving the correctness of algorithms. However, there are now many tools in existence that apply deductive verification to actual software. The early tools, such as SPARK [8], were labour intensive and required manual annotations to write pre- and post-conditions. Later tools, such as ESC/Java [45], use weakest precondition/strongest postcondition alongside Hoare's inductive rule for loops to generate assertion s, and then deployed independent theorem provers [45] or SMT solvers [11,43] to check the conditions.

Industry: SPARK [8] has been used in civil and military avionics, railway signalling and cryptographic solutions; Why3 has been used for proof of smart contracts [67]; Boogie [11] is developed and maintained by Microsoft, and has, amongst other things, been used for smart contract verification [80].

Running Example: The first thing we need to do is provide a loop invariant for the loop. We will use $(found \neq 0 \implies a[last] = target)$ (this is the same as assertion 3, and often invariants may be guessed from the properties we wish to prove). We then generate the verification conditions in Table 4, which correspond to the path from the start of the function to the invariant at the top of the loop, the path from the invariant around the loop once, and the path from the invariant exiting the loop via the loop condition. Assertions that cannot be proven mean either the specification is not met or that the invariant is too weak. However, there is no automatic technique that can tell the difference between the two in all cases. In this instance, it is possible to prove assertion 3 always holds but not that assertion 2 fails or assertion 1 always holds. But, if we make the invariant stronger, and use $(found \neq 0 \implies a[last] = target) \wedge i > 0$, we can now prove assertion 1 always holds as well.

3.6 Functional Verification

Functional verification comprises techniques that use mathematical reasoning to show equivalence between functional programs and constructive proofs. The result that it builds on is the connection between function application (β-reduction) in typed lambda calculus and modus ponens in intuistic logic:

$$\frac{t : A \quad \lambda x.E : A \to B}{E[x := t] : B} \qquad\qquad \frac{A \quad A \Rightarrow B}{B}$$

This can be seen as giving a logical character to programs; showing that a function $f : A \to B$ is well-typed is equivalent to proving that if t meets the precondition (is of type A), then $f(t)$ meets the postcondition (is of type B). This allows type checking and type inference algorithms to be used as verification tools.

This school of verification is a branch of functional programming as it is limited to programming languages and type systems that have an equivalence with a suitable logic. These languages tend to be functional as the logical equivalent of mutable state and pointers remain open research questions. As a result, functional verification is most effectively applied to *build* code that is correct by construction rather than to verifying code that already exists. Types play a similar role to annotations in deductive verification systems, giving the specification to be proven and the intermediate steps used to assist the verification tool / type checker. From this point of view, inductive invariants in loops are equivalent to type declarations for recursive functions and the repetition between proofs and programs found in deductive verification is avoided. However, the tight link between types and logic means that the specification must be expressible in the type system. It also requires a high level of skill as both the program and proof must be constructed simultaneously.

Culturally aligned with functional verification but with distinct foundations, there are a number of approaches to verification that use *Interactive Theorem Provers* (ITPs). The user constructs a mathematical proof that the program meets the specification and the ITP then checks this proof.

Industry: Despite the high skill level required, functional verification has been used for various projects with complex functional specifications that required non-trivial proof, e.g., the seL4 project [59] and the CompCert C Compiler [65].

Running Example: As our example program in Figure 3 is written in C we cannot directly demonstrate this style of verification. However, we can consider what would be needed to verify a functional implementation of the same system. To show that all of the array accesses were in bounds we would need an array type that included the length (i.e. (*array* 10) could be a valid type) and we would need an integer type that could express bounds, or a type inference algorithm that could determine them automatically.

4 Specifications

The specification is the set of criteria we check the program against. If you are building or maintaining a system then you need a specification – an understanding of what the system should do. Otherwise, you have no way of saying if the system is working as intended!

In this paper we are interested in specifications that are or can be formalised, i.e., expressed in a language with formal semantics.

There are many different kinds of specifications (“the program must be able to”, “the program must always”, “the program must never”, etc.) and the difficulty of verifying them can vary significantly. Tools and techniques for formal verification are often only applicable or are most suitable for certain kinds or parts of specifications. For example, showing that the program has a print feature is (hopefully!) fairly simple and testing may give sufficient evidence for the verification case. On the other hand, showing that there are no executions of the program with buffer over-runs is harder and will likely need software verification tools to achieve a reasonable degree of certainty.

4.1 Ways of Expressing Specifications

Which specifications can be used and how they are represented depends on the verification tool.

Builtins. One approach is to have a number of specifications built in to the tool and to have the user pick which one(s) they wish to verify against, for example, "no trace executes undefined behaviour". This approach is the easiest from the point of view of the user, setting the specification is ticking a few boxes or setting command-line flags. It is also convenient for tool developers as the verification tool can be specialised to handle the particular specifications supported. However, it is limited; if the tool does not support a relevant specification then it will be of little use, even if the core analysis that it is performing is relevant to the task.

Annotation. Another approach to specification is annotation. *Annotations* are statements in the program that describe a set of traces with reference to the location of the annotation. These may be written in comments, library calls (such as `assert`) or specific language constructs (such as pragmas). One way of classifying annotations is by how they describe the set of traces. They can refer to the state of the program when(ever) it reaches the annotation, for example giving constraints on values (`0 <= i && i < n`). These describe all of the traces that have the required state when they reach the annotation. They can refer to the future behaviour of the traces after they have reached the annotation, for example, termination. They can refer to the past behaviour of the traces before reaching the annotation, for example, taint (this parameter must not be influenced by user input). Implicit in the idea of annotations is a notion of reachability; annotations only apply when a trace reaches their location. This means that the verification tool must determine which parts of the program can be executed and so Rice's Theorem applies (see Sect. 2).

Annotations are more flexible than fixed specifications, they can be developed and maintained in parallel with the software and they can be used in a modular fashion and re-used along with the software. They can also be used to assist the verification tool by providing predicates that the tool can use for modular reasoning (see Sect. 3.5). Some tools such as Dafny [64] make a distinction between annotations that express specifications and those for assisting the proof. Describing specifications by annotations has some disadvantages. It interleaves the specification and implementation, meaning that they often have to be developed together or at least by teams who understand both aspects. They are also harder to review independently of the implementation.

External Objects. A third popular way of providing a specification is an external object, written in some formal language. Examples of this include providing another program as a specification and verifying the *equivalence* of the two (the specification should represent the same set of traces as the program) or providing a more abstract program as a specification and showing that the concrete program is a *refinement* of it (the traces in the program are a subset of the traces permitted by the specification). These approaches can be very useful if a reference model, protocol or implementation is available or in a hierarchical development approach where one language is used to create a series of progressively more detailed implementations by showing that each is a refinement of the previous one. One recent and promising direction in verification is to use *a previous version* of the software as the specification. This is *differential verification* [62] and allows us to check specifications about the *changes* between versions; "this modification does not introduce new bugs", "the change only affects a bounded amount of the program" or even "this patch definitely blocks a given exploit". Specifications as a separate object allow the most flexibility and reuse as well as giving a good separation between the development of the program and the specification. However, it may require significant extra development (differential verification is the notable exception to this requiring almost no extra effort) potentially as much as developing the program itself.

5 Using the Pyramid

The pyramid model allows us to classify and contrast techniques by which of the three key properties they guarantee and which they use computation to work towards. It is also a useful model for designing verification work-flows and selecting and evaluating appropriate tools.

For a particular project, there will only be a small (finite) number of programs and specifications of interest, thus Turing's result is not directly applicable. It would be possible (in theory) to create a verification tool that could achieve all three properties *for the programs and specifications in that project*. Developing such a project-specific tool is not financially viable for most organisations, so a process needs to be created using existing (or customised) tools. Given a particular program and specification, the key question is whether a reasonable amount of computation will reduce the missing attribute (false alarms, unexplored areas or human effort) by an acceptable amount. This question can only be answered by considering the wider context of the project; what role does the verification of the software play in the correctness, safety, security or performance of the system? What happens if the system is wrong? How much software already exists and how much can it be modified?

For example, if the verification is used to make a claim of code quality then using an over-approximate technique might be fine if the number of defects (including false alarms) is below the required threshold. An under-approximate technique might be suitable for a component with redundancy or fail-safe mechanisms as software defects will cause a loss of service rather than failure. Reducing the number of these is clearly beneficial but it is not necessary to remove all of them as low probability defects have a small impact. If defective software would cause a catastrophic system failure then a human-assisted technique might be most appropriate as a proof of correctness of the software can strengthen the system-wide verification case.

By prioritising the three attributes (automatic, no missed bugs, no false alarms) with respect to the project's goals and the need for software verification, the pyramid model can help select the right kind of tool.

5.1 Process

The human side of using software verification tools covers three aspects; developing the software, developing the specification and dealing with the missing third attribute. The first two of these are common to all approaches and are covered by both formal and non-formal development methodologies. The pyramid model can help inform the third area:

- If an over-approximate technique is used, then human effort will be required to deal with the output which will be a mix of false alarms and genuine defects. This was widely regarded as tedious but tractable.
  Unfortunately there is evidence [38] that even skilled developers are not able to reliably distinguish between false and real alarms.
- If an under-approximate analysis is used then any defects that are found are definitely real. Most tools of this kind will produce a trace or test-case that demonstrates the issue. These are often of considerable value for developers in fixing the problem [26]. For these approaches the missing attribute is 'no missed bugs' and so effort has to be put into increasing coverage. One way of achieving this is to create more fine-grained specifications; similar to unit tests. If checking the whole program leaves areas of the state-space unexplored, these can be used to increase the coverage. For example, if a monolithic under-approximate analysis does not check a particular function, the function can be checked independently. To do this requires a specification that includes the range of values of input variables over which the function is to be verified. The developer experience is likely to be similar to writing unit tests but using constraints to describe a space of possible inputs rather than fixed values.
- Finally if a human-assisted tool is used then the human effort in the process will be in producing the parts of the proof the tool is unable to directly infer. These will typically be pre and post-conditions for functions and loop invariants (see Sect. 3.5) or types (see Sect. 3.6). This requires developers with relevant training or experience.

5.2 Understanding Tool Evaluation

The pyramid model also helps understand how tools are evaluated in academia and industry. If tools are viewed as having two of the three attributes then evaluation is a question of measuring or approximating how each unit of computational effort reduces the number of missed bugs, false alarms or the amount of human-written proof across the set of all programs. As discussed earlier in this section, most tool users only care about the specific programs and specifications that they have. Combined with the obvious difficulty of running experiments over the (infinite) set of all programs and specifications means that most evaluations are conducted with *benchmarks*, sets of programs and specifications which are claimed to be representative.

Experimental evaluations of over-approximate tools tend to focus on the relative alarm rates or the relative difference between approximations given by different techniques. Underapproximate techniques tend to compare the speed at which different tools solve the same problem(s) (i.e., finding known bugs) or the number solved with given human effort. This is because measuring missed bugs is hard. Human-assisted tools use proxies to estimate the amount of effort needed, for example number of lines of proof, or ratio of lines of proof to lines of code, or the number of human-hours it takes to produce. Note that many of these tools take very different inputs, so it is very hard to compare directly and measuring effort/expertise is very subjective.

6 Conclusion

We have found the pyramid of verification to be an invaluable framework for classifying and choosing verification techniques, for teaching, and for bridging the gap between academics and potential users of verification tools. We hope that this paper enables the reader to do the same.

References

1. Coverity Scan: Static analysis. https://scan.coverity.com/. Accessed 10 Apr 2024
2. Cppcheck: A tool for static C/C++ code analysis. https://cppcheck.sourceforge.io/. Accessed 10 Apr 2024
3. CREST: Concolic test generation tool for C. https://www.burn.im/crest/. Accessed 20 July 2020
4. FindBugs. http://findbugs.sourceforge.net/. Accessed 22 July 2020
5. Fortify static code analyzer. https://www.opentext.com/products/fortify-static-code-analyzer. Accessed 10 Apr 2024
6. MALPAS software static analysis toolset. http://malpas-global.com/. Accessed 10 Apr 2024
7. PolySpace Code Prover. https://www.mathworks.com/products/polyspace-code-prover.html. Accessed 22 July 2020
8. SPARK. https://www.adacore.com/about-spark. Accessed 10 Apr 2024
9. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
10. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Burke, M., Soffa, M.L. (eds.) Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001, pp. 203–213. ACM (2001)
11. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17

12. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-662-07964-5
13. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
14. Biere, A.: Bounded model checking. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 457–481. IOS Press (2009)
15. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Irwin, M.J. (ed.) Proceedings of the 36th Conference on Design Automation, New Orleans, LA, USA, June 21-25, 1999, pp. 317–320. ACM Press (1999)
16. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
17. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10ˆ20 states and beyond. In: Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990, pp. 428–439. IEEE Computer Society (1990)
18. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp. 209–224. USENIX Association (2008)
19. Cadar, C., et al.: Symbolic execution for software testing in practice: preliminary assessment. In: ICSE, pp. 1066–1071. ACM (2011)
20. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
21. Cassé, H., Féraud, L., Rochange, C., Sainrat, P.: Using the abstract interpretation technique for static pointer analysis. SIGARCH Comput. Architect. News **27**(1), 47–50 (1999)
22. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
23. Cavada, R., et al.: The NUXMV symbolic model checker. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
24. Chen, D., Huang, R., Qu, B., Jiang, S.: Improving static analysis performance using rule-filtering technique. In: Reformat, M. (ed.) The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013, pp. 19–24. Knowledge Systems Institute Graduate School (2014)
25. Clarke, E., Kroening, D., Ouaknine, J., Strichman, O.: Completeness and complexity of bounded model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 85–96. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_9

26. Clarke, E., Veith, H.: Counterexamples revisited: principles, algorithms, applications. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 208–224. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39910-0_9
27. Cook, B.: Formal reasoning about the security of amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 38–47. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_3
28. Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from AWS data centers. Formal Methods Syst. Des. **57**(1), 34–52 (2021)
29. Coppa, E., D'Elia, D.C., Demetrescu, C.: Rethinking pointer reasoning in symbolic execution. In: Rosu, G., Penta, M.D., Nguyen, T.N. (eds.) Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, pp. 613–618. IEEE Computer Society (2017)
30. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM (1977)
31. Cousot, P., Cousot, R.: Static determination of dynamic properties of generalized type unions. In: Language Design for Reliable Software, pp. 77–94. ACM (1977)
32. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282. ACM Press (1979)
33. Cousot, P., et al.: The ASTREÉ analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_3
34. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
35. D'Abruzzo Pereira, J., Vieira, M.: On the use of open-source C/C++ static analysis tools in large projects. In: 2020 16th European Dependable Computing Conference (EDCC), pp. 97–102 (2020)
36. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453-457 (1975)
37. Dijkstra, E.W.: EWD 1308: What Led to "Notes on Structured Programming". In: Broy, M., Denert, E. (eds.) Software Pioneers, pp. 340–346. Springer, Heidelberg (2002). https://doi.org/10.1007/978-3-642-59412-0_19
38. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI, pp. 181–192. ACM (2012)
39. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using *k*-induction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 351–368. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_26
40. D'Silva, V.V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **27**(7), 1165–1178 (2008)
41. Ernst, M.D., et al.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
42. Ferdinand, C.: Worst case execution time prediction by static program analysis. In: IPDPS. IEEE Computer Society (2004)
43. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8

44. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45251-6_29
45. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI, pp. 234–245. ACM (2002)
46. Floyd, R.W.: Assigning meanings to programs. In: Colburn, T.R., Fetzer, J.H., Rankin, T.L. (eds) Program Verification. Studies in Cognitive Systems, vol. 14. Springer, Dordrecht (1993). https://doi.org/10.1007/978-94-011-1793-7_4
47. Gibson-Robinson, T.: FDR3: the future of CSP model checking. In: Welch, P.H., Barnes, F.R.M., Broenink, J.F., Chalmers, K., Pedersen, J.B., Sampson, A.T. (eds.) 35th Communicating Process Architectures, CPA 2013, Edinburgh, Scotland, UK, August 25, 2013, pp. 321–322. Open Channel Publishing Ltd. (2013)
48. Godefroid, P.: Software model checking improving security of a billion computers. In: Păsăreanu, C.S. (ed.) SPIN 2009. LNCS, vol. 5578, pp. 1–1. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02652-2_1
49. Gopinath, R., Jensen, C., Groce, A.: Code coverage for suite evaluation by developers. In: Jalote, P., Briand, L.C., van der Hoek, A. (eds.) 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India, May 31 - June 07, 2014, pp. 72–82. ACM (2014)
50. Havelund, K.: Java PathFinder a translator from Java to Promela. In: Dams, D., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 152–152. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48234-2_11
51. Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST software verification system. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 25–26. Springer, Heidelberg (2005). https://doi.org/10.1007/11537328_4
52. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
53. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
54. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), 21:1–21:54 (2009)
55. Johnson, S.C.: Lint, a C program checker, pp. 78–1273 (1978)
56. Kästner, D., Wilhelm, R., Ferdinand, C.: Abstract interpretation in industry - experience and lessons learned. In: In: Hermenegildo, M.V., Morales, J.F. (eds) Static Analysis. SAS 2023. Lecture Notes in Computer Science, vol 14284. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-44245-2_2
57. Kettl, M., Lemberger, T.: The static analyzer infer in SV-COMP (competition contribution). In: TACAS 2022. LNCS, vol. 13244, pp. 451–456. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_30
58. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
59. Klein, G., Elphinstone, K., et al.: seL4: formal verification of an OS kernel. In: SOSP, pp. 207–220. ACM (2009)
60. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
61. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China, June 11 - 16, 2012, pp. 193–204. ACM (2012)

62. Lahiri, S.K., Vaswani, K., Hoare, C.A.R.: Differential static analysis: opportunities, applications, and challenges. In: FoSER, pp. 201–204. ACM (2010)
63. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis and transformation. In: CGO, pp. 75–88. IEEE Computer Society (2004)
64. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
65. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
66. Logozzo, F.: Practical specification and verification with code contracts. In: HILT, pp. 7–8. ACM (2013)
67. Nehaï, Z., Bobot, F.: Deductive proof of industrial smart contracts using Why3. In: Sekerinski, E., et al. (eds.) FM 2019. LNCS, vol. 12232, pp. 299–311. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54994-7_22
68. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
69. Norell, U.: Dependently typed programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04652-0_5
70. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_217
71. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M.: Chapter six - mutation testing advances: an analysis and survey. Adv. Comput. **112**, 275–378 (2019)
72. Pasareanu, C.S., et al.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: ISSTA, pp. 15–26. ACM (2008)
73. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Trans. Am. Math. Soc. **74**, 358–366 (1953)
74. Schmidt, D.A.: Data flow analysis is model checking of abstract interpretations. In: MacQueen, D.B., Cardelli, L. (eds.) POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19–21, 1998, pp. 38–48. ACM (1998)
75. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Successful use of incremental BMC in the automotive industry. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 62–77. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19458-5_5
76. Shen, H., Fang, J., Zhao, J.: EFindBugs: effective error ranking for findBugs. In: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011, pp. 299–308. IEEE Computer Society (2011)
77. Tillmann, N., de Halleux, J.: Pex–white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_10
78. Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. Proc. London Math. Soc. **s2-42**(1), 230–265 (1937)
79. Vernier-Mounier, I.: Symbolic executions of symmetrical parallel programs. In: 4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96), January 24-26, 1996, Portugal, pp. 327–335. IEEE Computer Society (1996)

80. Wang, Y., et al.: Formal verification of workflow policies for smart contracts in Azure Blockchain. In: Chakraborty, S., Navas, J.A. (eds.) VSTTE 2019. LNCS, vol. 12031, pp. 87–106. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41600-3_7
81. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. In: Deusen, M.S.V., Galil, Z., Reid, B.K. (eds.) Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985, pp. 291–299. ACM Press (1985)
82. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. ACM SIGSOFT Softw. Eng. Notes **30**(2), 1–36 (2005)
83. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Advancing Quantum Computing with Formal Methods

Arend-Jan Quist, Jingyi Mei, Tim Coopmans(✉), and Alfons Laarman

Leiden University, Leiden, The Netherlands
{a.quist,j.mei,t.j.coopmans,a.w.laarman}@liacs.leidenuniv.nl

Abstract. This tutorial introduces quantum computing with a focus on the applicability of formal methods in this relatively new domain. We describe quantum circuits and convey an understanding of their inherent combinatorial nature and the exponential blow-up that makes them hard to analyze. Then, we show how weighted model counting (#SAT) can be used to solve hard analysis tasks for quantum circuits.

This tutorial is aimed at everyone in the formal methods community with an interest in quantum computing. Familiarity with quantum computing is not required, but basic linear algebra knowledge (particularly matrix multiplication and basis vectors) is a prerequisite. The goal of the tutorial is to inspire the community to advance the development of quantum computing with formal methods.

1 Introduction

"Nature isn't classical, (...), and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy"

Richard Feynman [11]

Renowned physicist and Nobel prize winner Richard Feynman is said to be the first to coin the idea of a 'quantum computer' by inverting the difficulty he encountered when solving equations in quantum mechanics [11]. He proposed to make nature's complexity work for us, instead of trying to calculate nature's behavior using classical tools —which Feynman noticed requires performing exponentially many calculations. Fast forward to today, and the first quantum computers have been built arguably showing first signs of 'quantum advantage' [3].

We see that powerful techniques from formal methods are ideally suited to tackle some of the crucial problems on the road towards a full-scale quantum

A.-J. Quist, J. Mei and T. Coopmans–These authors contributed equally.

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 420–446, 2025.
https://doi.org/10.1007/978-3-031-71177-0_25

advantage. This tutorial, therefore, provides a crash course into quantum computing, specifically geared towards a formal methods audience. Our goal is to inform the community of the challenges in handling quantum computations and point them in the right direction to apply their own favorite techniques on open problems in the field of quantum computing (Sect. 4).

For inspiration, we show how to formulate the semantics of quantum circuits using #SAT. At the same time, we keep the quantum background to an absolute minimum by introducing quantum computing as a minor extension of reversible and probabilistic computation,[1] and gradually introducing the required notation.

Nevertheless, our basic exposition manages to touch upon quantum algorithms, such as Grover search [15], by briefly discussing how the Boolean satisfiability problem can be incorporated into a quantum circuit as an oracle. While the satisfying assignments can then be found in the so-called *probability amplitudes* of the quantum state that is computed by the circuit, it is hard to extract such information using *measurements*, the only method to extract physical information from a quantum system (which is unfortunately rather crude). Such an example should inspire listeners to think about how to extract the solution to the satisfiability problem using measurements, which is the difficulty of coming up with so-desired quantum algorithms that outperform their classical alternatives.[2]

Our exposition here focuses on the task of (classically) simulating quantum circuits, which is formally defined in the background section. This ensures that the audience becomes familiar with the semantics of quantum circuits and algorithms (which can be formulated as uniform families of quantum circuits). At the very end, we show, based on related works, how the discussed encodings in #SAT can also be used to solve various other tasks, inter alia: (quantum circuit) equivalence checking, synthesis, optimization and quantum Hoare logic checking. We also point the audience in the direction of the current obstacles on the road to quantum supremacy, such as, circuit optimization and quantum error correction (which will crucially realize the ideal quantum circuit model introduced here). Here, we even point out connections beyond quantum computing since it is easy to show that progress in handling quantum circuits translates to progress in solving important and hard problems in quantum mechanics, like simulation of many-body physics and finding ground states.

[1] Surprisingly, not even complex numbers are needed, as the (classical) reversible Toffoli gate, which is universal for reversible computing, requires only an additional Hadamard (Walsh-Hadamard/Reed-Muller transform) gate to yield a universal gate set for quantum computation [1,34].

[2] While there is, for instance, an efficient quantum algorithm for factoring (Shor's [35]), we do not know for sure whether factoring is hard classically. So a true *separation* between the classical and quantum complexity classes is still open (formalized as the $\mathsf{BPP} = \mathsf{BQP}$ question).

2 Quantum Computing for Computer Scientists

"Quantum computing becomes easy once you take the physics out of it"

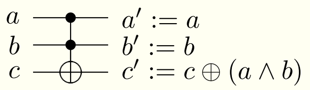
https://scottaaronson.blog

This section introduces quantum circuits as a generalization of (classical) reversible and probabilistic computing. Having provided the reader with an understanding of these classical building blocks, we then explore the semantics of quantum circuits. We finish by highlighting the crucial aspects of a quantum circuit that make formal methods amenable to it.

2.1 From Reversible, via Probabilistic, to Quantum Circuits

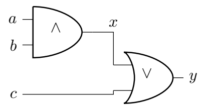
Each (irreversible) Boolean circuit can be written as a reversible circuit. One way to do this is using circuits with only the so-called Toffoli gate [37] (see the yellow box below).

Toffoli gate. The three-bit Toffoli gate, shown below, is both reversible and classically universal. It takes three bits a, b, c as inputs and outputs $a, b, c \oplus ab$. Informally, it only flips the c bit when both a and b are true (leaving a and b intact).

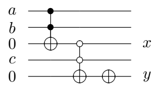


Applying a Toffoli twice will compute $c \oplus ab \oplus (a \wedge b) = c$, thus reversing the original computation by 'uncomputing' the result. Furthermore, by setting $a = b = 1$, the c bit is always flipped so that we can realize a NOT gate ($\neg$) as syntactic sugar (drawn as $\oplus$ in a circuit). Toffoli is classically universal, which follows from the fact that the gate set $\{\wedge, \neg\}$ is universal and that Toffoli implements an AND gate ($\wedge$). By setting $a = 1$, we obtain a (singly-)controlled-NOT, or CNOT gate (drawn as $\begin{smallmatrix}\bullet\\ \oplus\end{smallmatrix}$), which flips the value of c only when b is set. As additional syntactic sugar, we use the negated control $\circ$, which is a control $\bullet$ surrounded by negations $\oplus\bullet\oplus$. It checks whether the input is false, returning it to its original state.

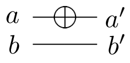
As an example of writing each Boolean circuit as a reversible circuit, consider circuit (i) on input $a, b, c \in \{0, 1\}$ below: We construct a reversible circuit by storing intermediate value x and the result y separately, adding wires for these variables. This leads to the reversible circuit (ii) containing NOT gates (drawn as $\oplus$ indicating a bit flip without control) and Toffoli gates (a bit flip $\oplus$ with two controls $\bullet$ or $\circ$).



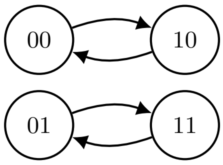
(i)



(ii)



(iii)

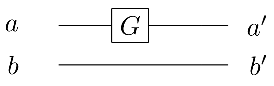


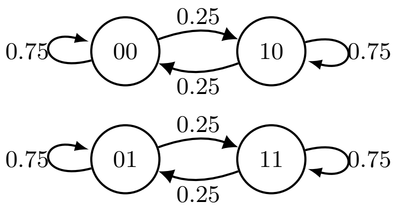
(iv)

Next, let us visualize the action of a classical reversible circuit. In (iii) above, we use a circuit on two bits a, b and give in (iv) an automaton of the four possible states on two bits. An arrow from state (a, b) to (a', b') indicates that the circuit maps (a, b) to (a', b'). (We use a circuit on two bits instead of five as in (ii) to avoid too large automata for easy readability.)

Exercise: Create a reversible circuit for the Boolean formula $((a \wedge b) \vee c) \wedge d$. Then extend the circuit to uncompute all wires but the result.

2.1.1 Probabilistic Classical (reversible) Computing with Linear Algebra To move closer towards quantum circuits, we will now see how the automaton changes when the logical gates used are probabilistic. Specifically, in example (iii) above, we replace the NOT gate ($\oplus$) with a probabilistic gate G, which does nothing with probability 75% and applies a NOT with 25%, resulting in (v) below. Then the circuit is modeled by a Markov chain, and the transition arrows in the automaton (vi) are labeled with the probabilities of mapping the input state (a, b) to output state (a', b') [16].

(v)


(vi)


A Markov chain is typically analyzed by associating each state (a, b) (abbreviated 'ab') in the automaton to a basis vector and writing down the transition probabilities as an adjacency matrix M of the automaton above:

$$\text{`00'} \equiv \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \text{`01'} \equiv \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \text{`10'} \equiv \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \text{`11'} \equiv \begin{bmatrix}0\\0\\0\\1\end{bmatrix}, \ M = \begin{bmatrix}0.75 & 0 & 0.25 & 0\\ 0 & 0.75 & 0 & 0.25\\ 0.25 & 0 & 0.75 & 0\\ 0 & 0.25 & 0 & 0.75\end{bmatrix} \quad (1)$$

Next, the probability distribution over the output states (a', b') on input (a, b) is computed using matrix-vector multiplication. For example, on input '00' ($a = 0, b = 0$), the output is

$$M \cdot \text{`00'} = \begin{bmatrix} 0.75 & 0 & 0.25 & 0 \\ 0 & 0.75 & 0 & 0.25 \\ 0.25 & 0 & 0.75 & 0 \\ 0 & 0.25 & 0 & 0.75 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.75 \\ 0 \\ 0.25 \\ 0 \end{bmatrix} = 0.75 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + 0.25 \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix},$$

that is, '00' is mapped to itself with probability 75% and to '10' with probability 25%, as was already revealed by the Markov chain.

The advantage of using the transition matrix instead of writing down the Markov chain is that the input need not be a single bitstring itself but can also be a probability distribution over bitstrings. In this more general case, **matrix-vector multiplication correctly propagates the probabilities.** To see this, we first directly compute the probabilities of the output bitstrings for the circuit in (v) using the Markov chain. We use the example of an input state that is either set to the bitstring '00' with probability 0.4 or to the bitstring '10' with probability 0.6. In the first case, the output of the circuit in (v) is '00' with a probability of 0.75 and '10' with a probability of 0.25; in the second case, it is '00' ('10') with a probability of 0.25 (0.75). Aggregating these two, the output is '00' with probability $0.4 \cdot 0.75 + 0.6 \cdot 0.25 = 0.45$ and '10' with probability $0.4 \cdot 0.25 + 0.6 \cdot 0.75 = 0.55$. Equivalently, we obtain this result using linear algebra. The input, which represents '00' with probability 0.4 and '10' with probability 0.6, is the vector $\boldsymbol{v} = \begin{bmatrix} 0.4 \; 0 \; 0.6 \; 0 \end{bmatrix}^\top$ (where the transpose $(.)^\top$ turns a row vector into a column vector). The output probability distribution is computed by applying the transition matrix to the input vector $\boldsymbol{v}$:

$$M \cdot \boldsymbol{v} = \begin{bmatrix} 0.4 \cdot 0.75 + 0.6 \cdot 0.25 \\ 0 \\ 0.4 \cdot 0.25 + 0.6 \cdot 0.75 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.45 \\ 0 \\ 0.55 \\ 0 \end{bmatrix},$$

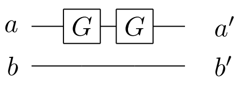
representing 45% in state '00' and 55% in state '10', which is precisely the same result we obtained before! Indeed, matrix-vector multiplication correctly propagates the probabilities.

We note that because the input vector represents a probability distribution over bitstrings, its entries should be probabilities, i.e., nonnegative values between zero and 1, and the vector should be *normalized* (all its entries should sum to 1). Also, each column in the transition matrix has the same property, as the sum of outgoing-edge labels of a node in the automaton constitute a probability distribution themselves too.

Key message: An n-bit reversible circuit with probabilistic gates is represented by a transition matrix. If the input to the circuit is a probability distribution over bitstrings, it can be represented by a 2^n-length vector containing the probabilities. Furthermore, the action of the circuit is computed by applying the transition matrix to the input vector.

Finally, in order to compute the probability of ending in a certain state, say '10', after starting in $\boldsymbol{v}$, we can compute the following: $\text{`10'}^\top \cdot M \cdot \boldsymbol{v} \equiv \begin{bmatrix}0 \; 0 \; 1 \; 0\end{bmatrix} \cdot M \cdot \boldsymbol{v} = 0.55$. Note the use of a row vector ($\text{`10'}^\top$) and a column vector ($\boldsymbol{v}$).

Sequential Composition of Circuits Using Matrix Multiplication. Suppose that the input probability distribution over bitstrings $\boldsymbol{v}$ is inputted to circuit A, yielding the output vector $\boldsymbol{w} = M_A\boldsymbol{v}$, where M_A is the transition matrix of A. Suppose furthermore that we pass input $\boldsymbol{w}$ to another circuit B. Then the probability vector representing B's output is $M_B\boldsymbol{w} = M_B(M_A\boldsymbol{v})$, which can also be written as $(M_B \cdot M_A)\boldsymbol{v}$, where $\cdot$ denotes matrix multiplication. Since this holds for any input vector $\boldsymbol{v}$, we observe that a circuit C, consisting of first performing A followed by B, has a transition matrix $M_B \cdot M_A$. That is, the sequential composition of gates, and thus of circuits, corresponds to the matrix multiplication of their transition matrices. For example, in (v) the transition matrix is denoted as M and G is applied to the top bit a once, so applying G twice instead yields M^2:



$$M^2 = \begin{bmatrix} 0.75 & 0 & 0.25 & 0 \\ 0 & 0.75 & 0 & 0.25 \\ 0.25 & 0 & 0.75 & 0 \\ 0 & 0.25 & 0 & 0.75 \end{bmatrix}^2 = \begin{bmatrix} 0.625 & 0 & 0.375 & 0 \\ 0 & 0.625 & 0 & 0.625 \\ 0.375 & 0 & 0.375 & 0 \\ 0 & 0.375 & 0 & 0.625 \end{bmatrix}$$

Parallel Composition of Circuits Using the Kronecker Product. Above, we found that the sequential composition of circuits corresponds to the matrix-multiplication product of their transition matrices. For parallel composition, where one stacks two circuits 'on top' of each other, we need the Kronecker product of matrices.

Kronecker product. The Kronecker product of matrices A and B performs a giant case distinction: for each entry a of A, we create a copy of all possible entries b of B, and the resulting matrix contains the values $a \cdot b$. For example, the transition matrices on a single bit

$$A = \begin{bmatrix} 0.1 & 0.2 \\ 0.9 & 0.8 \end{bmatrix}, \quad B = \begin{bmatrix} 0.3 & 0.4 \\ 0.7 & 0.6 \end{bmatrix}$$

is

$$A \otimes B = \begin{bmatrix} 0.1 \cdot B & 0.2 \cdot B \\ 0.9 \cdot B & 0.8 \cdot B \end{bmatrix} = \begin{bmatrix} 0.1 \cdot 0.3 & 0.1 \cdot 0.4 & 0.2 \cdot 0.3 & 0.2 \cdot 0.4 \\ 0.1 \cdot 0.7 & 0.1 \cdot 0.6 & 0.2 \cdot 0.7 & 0.2 \cdot 0.6 \\ 0.9 \cdot 0.3 & 0.9 \cdot 0.4 & 0.8 \cdot 0.3 & 0.8 \cdot 0.4 \\ 0.9 \cdot 0.7 & \underline{0.9 \cdot 0.6} & 0.8 \cdot 0.7 & 0.8 \cdot 0.6 \end{bmatrix}.$$

For example, the entry of $A \otimes B$ at column 2 ($01 = 0_A 1_B$ in binary) and row 4 ($1_A 1_B$), which is underlined above, contains the probability $0.9 \cdot 0.6$ which is the probability that *both* A maps bit 0 to 1 (which happens with probability 0.9) as well as B maps 1 to 1 (which happens with probability 0.6).

Formally, the Kronecker product on two general matrices (not necessarily transition matrices) acts as follows: given $r_A \times c_A$ matrix A and $r_B \times c_B$ matrix B, the $r_A r_B \times c_A c_B$ matrix $A \otimes B$ is

$$A \otimes B = \begin{bmatrix} A_{00}B & A_{01}B & \dots & A_{0c_A}B \\ A_{10}B & A_{11}B & \dots & A_{1c_A}B \\ \vdots & \vdots & \ddots & \\ A_{r_A0}B & A_{r_A1}B & \dots & A_{r_Ac_A}B \end{bmatrix}.$$

The product behavior of the Kronecker product of two matrices A and B can be interpreted as choosing an entry a from A and an entry b from B and multiplying them (see the yellow box above). When A and B are probability vectors of the individual systems, their Kronecker product yields precisely the vector on the combined system, where the values $a \cdot b$ are the occurrence probabilities of these automaton states/unit vectors. The same interpretation shows that the parallel composition of two gates is also given by their Kronecker product. For example, the transition matrix for G in (iii) (which acts on a single bit) is given by $M_G = \left[\begin{smallmatrix} 0.75 & 0.25 \\ 0.25 & 0.75 \end{smallmatrix}\right]$, the transition matrix on the second bit (doing nothing) is the identity matrix $I = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$; hence we may write $M_G \otimes I$ for the transition matrix on the two bits a, b.

Exercise: Using the definition of the Kronecker product sign $\otimes$ (see the yellow box above), verify that $M_G \otimes I = M$ from Eq. 1.

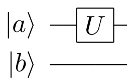
Exercise: For a single bit, the two basis vectors are '0' $\equiv \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$ and '1'$\equiv \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$. Compute the 4 possible Kronecker products between these states ('0' $\otimes$ '0', '0 $\otimes$ '1', etc.) and verify that these are precisely the four basis vectors for two bits from Eq. 1.

Key message: Composing probabilistic circuits sequentially corresponds to multiplying their transition matrices, while their parallel composition is represented by the Kronecker product of the transition matrices.

2.1.2 Quantum Computations with Linear Algebra Now let us move to quantum bits or qubits. *Here, we limit the presentation to a simplified model of quantum computing using solely real numbers. This model is universal for quantum computing* [1] *and suffices to explain all aspects of quantum computing. Nevertheless, a reader should keep in mind that the usual quantum computing model contains complex numbers and, therefore, comprises the subtle differences noted below (also see reading material referenced at the end of Sect.* 4*).*

The state of two qubits is a generalization of a probability distribution over the four bitstrings, whose vectors, e.g., for '00', '01', '10' and '11' in Eq. 1, are referred to as the *computational-basis states* as they form a basis of the space of 4-dimensional vectors, and written as $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ (so-called Dirac notation). Just like in the Markov chain case above, the state on n quantum bits is represented by a vector of length 2^n. However, in contrast to the Markov chain case, the entries of the vector are not probabilities (which sum up to 1) but are so-called *probability amplitudes*, or simply *amplitudes*. So to represent a quantum state, the vector should now be normalized by letting the *squares*[3] of these amplitudes sum up to 1. Instead of a transition matrix, which contains probabilities such that the entries in each column add up to 1, the gate in the circuit is given by a *unitary matrix*: in a unitary matrix, the sum of *squares* of the entries (amplitudes) in a column add up to 1.[4] Sequential and parallel composition of gates, identically to the case of probabilistic computing in Sect. 2.1.1, correspond to matrix multiplication and Kronecker product, respectively.

An example is the circuit below: the input is computational-basis state $|a\rangle \otimes |b\rangle = |ab\rangle$ for $a, b \in \{0, 1\}$, and the single-qubit quantum gate U in the circuit yields the two-qubit action M' (to see why $M' = U \otimes I$, compare with $M_G \otimes I$ in Sect. 2.1.1):



$$U = \begin{bmatrix} \sqrt{0.75} & \sqrt{0.25} \\ \sqrt{0.25} & -\sqrt{0.75} \end{bmatrix} \qquad M' = U \otimes I = \begin{bmatrix} \sqrt{0.75} & 0 & \sqrt{0.25} & 0 \\ 0 & \sqrt{0.75} & 0 & \sqrt{0.25} \\ \sqrt{0.25} & 0 & -\sqrt{0.75} & 0 \\ 0 & \sqrt{0.25} & 0 & -\sqrt{0.75} \end{bmatrix}$$

For example, if the input state is $|11\rangle \equiv \begin{bmatrix}0,0,0,1\end{bmatrix}^\top$, then the output state is $(U \otimes I) \cdot |11\rangle = \begin{bmatrix}0, \sqrt{0.25}, 0, -\sqrt{0.75}\end{bmatrix}^\top = \sqrt{0.25}|01\rangle - \sqrt{0.75}|11\rangle$. By squaring the amplitudes $\sqrt{0.25}$ and $\sqrt{0.75}$, we regain the interpretation as probability

[3] Or modulus square in the general case with complex numbers.

[4] The rows of a unitary matrix are also orthogonal; see Sect. 2.2 for full definition.

distribution over the states $|01\rangle$ and $|11\rangle$. *However*, the fact that the amplitudes can be *negative* (and complex numbers in general) is crucial: as we will see later, these give rise to *interference*, where amplitudes are canceled or amplified, a feature that is not present in probabilistic computing.

Key message: quantum computing is a generalization of probabilistic computing.
It will be useful to read a(n exponential-length) state vector as a probability (amplitude) distribution over classical (computational basis) states:

$$\begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \vdots \\ \alpha_{2^n-2} \\ \alpha_{2^n-1} \end{bmatrix} = \begin{bmatrix} \alpha_{0\ldots00} \\ \alpha_{0\ldots01} \\ \vdots \\ \vdots \\ \alpha_{1\ldots10} \\ \alpha_{1\ldots11} \end{bmatrix} \begin{array}{l} \rightarrow \text{ probability amplitude of } |0\ldots00\rangle \\ \rightarrow \text{ probability amplitude of } |0\ldots01\rangle \\ \\ \\ \rightarrow \text{ probability amplitude of } |1\ldots10\rangle \\ \rightarrow \text{ probability amplitude of } |1\ldots11\rangle \end{array}$$

2.2 Building Blocks of Quantum Circuits

In the previous section, we gave an example of a quantum state as a generalization of a probability distribution over bitstrings. Here we give the full definition of a general quantum state (state of a collection of qubits), and give the building blocks of quantum-computer circuits: the quantum analogs of logical *gates*, and *measurements* as a means to extract information from a quantum state. (For a full introduction, see [24].)

2.2.1 Quantum Bits As we have seen before, a quantum state on n qubits is a column vector of length 2^n, which constitutes a probability distribution over the length-n bitstrings $b \in \{0,1\}^n$ as the vector is a *linear combination* $\sum_{b\in\{0,1\}^n} \alpha_b |b\rangle$ over the corresponding computational-basis states $|b\rangle$ (i.e. the length-2^n vector with entry 1 at position b and 0 everywhere else). (The quantum-computing jargon is *superposition* instead of linear combination.) The vector entries (amplitudes) α_b are complex numbers in general. In this paper, we will always consider real vector entries. The square of the amplitude determines the probability. For example, the following vectors

$$\begin{bmatrix} -\sqrt{0.75} \\ \sqrt{0.25} \end{bmatrix}, \quad \begin{bmatrix} -\sqrt{0.75} \\ -\sqrt{0.25} \end{bmatrix}, \quad \begin{bmatrix} \sqrt{0.75} \\ \sqrt{0.25} \end{bmatrix} \tag{2}$$

all give rise to the probabilities 0.75 and 0.25 over the states $|0\rangle$ and $|1\rangle$, respectively. If the amplitudes α_b of a vector $\boldsymbol{v}$ are real numbers, then $\sqrt{\sum_{b\in\{0,1\}^n} \alpha_b^2}$

indicates the vector's length (for complex numbers, we would also have to take the modulus $|\alpha_b|$ instead of α_b). Since the values $|\alpha_b|^2$ form a probability distribution, we find that:

Key message: an n-qubit state is a vector of 2^n complex numbers with norm 1. (In this paper, we only consider real vectors.)

Example 1. The vectors $|\phi\rangle, |\psi\rangle$ and $|\eta\rangle$ are quantum states on 2, 2 and 3 qubits, respectively:

$$|\phi\rangle \triangleq \frac{1}{\sqrt{1^2+2^2+3^2+\sqrt{17}^2}} \cdot \begin{bmatrix} 1 \\ 2 \\ -3 \\ \sqrt{17} \end{bmatrix} = \frac{1}{\sqrt{31}} \left(1 \cdot |00\rangle + 2 \cdot |01\rangle - 3 \cdot |10\rangle + \sqrt{17}|11\rangle\right)$$

$$|\psi\rangle \triangleq \frac{1}{2} \cdot \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}, \qquad |\eta\rangle \triangleq \frac{1}{\sqrt{2}} \cdot \left[1,0,0,0,0,0,0,1\right]^\top = \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|111\rangle.$$

◇

Consider a set of six qubits and suppose that the first two are jointly in some arbitrary state $|\phi\rangle$ (a vector of length $2^2 = 4$) and the remaining four in some state $|\psi\rangle$ (a vector of length $2^4 = 16$). Identically to the case of probabilistic computing (Sect. 2.1.1), we use the Kronecker product $\otimes$ to find the joint state on the six qubits, which is $|\phi\rangle \otimes |\psi\rangle$ (a vector of length $2^2 \cdot 2^4 = 2^6$).

Exercise: Show that the computational-basis states $|b\rangle$ for $b = (b_1, b_2, \ldots, b_n) \in \{0,1\}^n$ can be written as $|b\rangle = |b_1\rangle \otimes |b_2\rangle \otimes \cdots \otimes |b_n\rangle$.

2.2.2 Manipulating Quantum States Using Gates A quantum gate on n qubits is represented by a 2^n by 2^n unitary matrix U (unitarity defined below). A quantum state $|\phi\rangle$ is updated by a unitary matrix as $U \cdot |\phi\rangle$ where $\cdot$ denotes matrix-vector multiplication.

Recall from Sect. 2.1 that the three-(qu)bit *CCNOT* gate (or doubly control-NOT, or Toffoli gate) is universal for classical computing, while the *NOT* and *CNOT* gates can be added as syntactic sugar. Below we give their matrices. To obtain a universal gate set for quantum computing, we merely need to extend this gate set with the single-qubit *Hadamard gate*: $H \triangleq \frac{1}{\sqrt{2}} \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$. Example 2 illustrates its function: realizing superpositions.

$$NOT \triangleq \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad CNOT \triangleq \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad CCNOT \triangleq \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Example 2. Applying the H gate to the state $|0\rangle$ or $|1\rangle$, we obtain

$$H \cdot |0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle),$$
$$H \cdot |1\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

◇

A square matrix U is unitary if it is invertible and its inverse is $U^\dagger$, where $U^\dagger$ is found by transposing U for real matrices.[5] Unitarity ensures that the matrix is norm-preserving, thus guaranteeing that the output quantum state has norm 1 if the input state does so too.[6] As unitary matrices are reversible, we directly see that all quantum gates are also reversible. This relates quantum computing to reversible computing. The Hadamard gate and NOT gate have adjoint operators $H^\dagger = \frac{1}{\sqrt{2}} \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right] = H$ and $NOT^\dagger = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right] = NOT$; it is not hard to check, indeed, that $H^\dagger \cdot H = NOT^\dagger \cdot NOT = I = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$.

To apply a single-qubit gate to one qubit of a state on $n > 1$ qubits, one uses the Kronecker product (see Sect. 2.1) for 'padding' the gate with an identity matrix I for each other qubit, i.e. the matrices which leave any vector unchanged. For example, applying H to the first qubit of $|010\rangle$ is computed as $(H \otimes I \otimes I)|010\rangle$.

Exercise: Compute the unitary matrix $H \otimes I \otimes I$. Remark that it has dimensions $2^3 \times 2^3$, as is needed for being able to apply it to a 3-qubit state.

[5] The adjoint $U^\dagger$ of a complex matrix U is found by transposing U, followed by replacing each matrix entry $u + wi$ with its complex conjugate $u - wi$. As we use only real matrices in this paper, one can always think of the adjoint as the transpose. The adjoint notation $U^\dagger$ is used in this paper, in favor of the inverse U^{-1}, as this is common in the quantum community.

[6] Quantum states should have norm 1 to be physically meaningful since the probabilites over the computational-basis states should sum up to 1. Further, quantum gates need to be unitary to obey the energy conservation law, due to quantum mechanical properties following from the famous Schödinger equation.

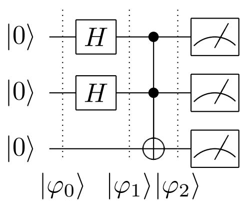
Finally, we remark that sequential composition of gates is represented by matrix multiplication of their unitaries, similar to matrix multiplication in the Markov chain case in Sect. 2.1: applying first U and then V to $|\phi\rangle$ yields the state $V \cdot (U \cdot |\phi\rangle) = (V \cdot U)|\phi\rangle$.

2.2.3 Measurement Measurement enables one to extract classical information (bits) from a quantum state. The theory of quantum measurement allows many ways to do this and here we only focus on a common one: measuring all qubits in the computational basis. Given an n-qubit quantum state $\sum_{b\in\{0,1\}^n} \alpha_b|b\rangle$, a measurement (on all qubits) is a probabilistic operation that returns one of the values $b \in \{0,1\}^n$, called the *measurement outcome*. The value b is returned with probability $(\alpha_b)^2$. (Or $|\alpha_b|^2$ in the case of complex amplitudes.) For example, the probability of measuring $00\ldots0$ is $(\alpha_{00\ldots0})^2$. Since each quantum state vector has unit norm, these probabilities add up to 1, as desired.

Example 3. Consider the state $\left[\frac{1}{2}, 0, \frac{1}{2}, 0, \frac{1}{2}, 0, 0, \frac{1}{2}\right]^\top = \frac{1}{2}|000\rangle + \frac{1}{2}|010\rangle + \frac{1}{2}|100\rangle + \frac{1}{2}|111\rangle$. Upon measurement, the probability to obtain measurement outcome 000 is $(\alpha_{000})^2 = \left(\frac{1}{2}\right)^2 = \frac{1}{4}$ and the probability to obtain measurement outcome 001 is $(\alpha_{001})^2 = 0^2 = 0$. ◇

2.2.4 Quantum Circuit A quantum circuit is composed of qubits represented by horizontal lines (wires) and quantum gates represented by boxes, with each gate acting on one or more qubits. The circuit is finished with a measurement. We will always let the input state be $|0\rangle \otimes |0\rangle \otimes \cdots \otimes |0\rangle$ (which is usually written as $|0\rangle^{\otimes n}$ or $|00\ldots0\rangle$), and the output state is obtained after the sequential application of the circuit's gates. All gates can be combined with matrix multiplication into a single unitary operator describing the entire circuit.

Example 4. In the following circuit, we apply a Hadamard to the first two qubits of $|000\rangle$. Then we apply a Toffoli gate ($CCNOT$) to the three qubits, followed by an all-qubit measurement. The intermediate states are as follows; see Example 3 for evaluating measurement.



$$|\varphi_0\rangle = |000\rangle$$
$$|\varphi_1\rangle = (H^{\otimes 2} \otimes I)|\varphi_0\rangle = \frac{1}{2}(|000\rangle + |010\rangle + |100\rangle + |110\rangle)$$
$$|\varphi_2\rangle = CCNOT|\varphi_1\rangle = \frac{1}{2}(|000\rangle + |010\rangle + |100\rangle + |111\rangle)$$

◇

We emphasize that quantum circuits are reversible: each gate, and hence each circuit, has the same number of input and output qubits.

Suppose that we have an n-qubit circuit C initialized to the all-zero state $|00\ldots0\rangle$. Computing the probability of outcome $b \in \{0,1\}^n$ can now be written as follows: $|\langle b|C|00\ldots0\rangle|^2$. Here, $\langle b|$ is a row vector of the computational basis state $|b\rangle$, so $\langle b|C|00\ldots0\rangle$ can be seen as a product of a row vector, matrix and column vector, resulting in a scalar.

Example 5. Let C be the circuit from Example 4. Then $C|000\rangle$ equals $\frac{1}{2}|000\rangle + \frac{1}{2}|010\rangle + \frac{1}{2}|100\rangle + \frac{1}{2}|111\rangle = \left[\frac{1}{2}, 0, \frac{1}{2}, 0, \frac{1}{2}, 0, 0, \frac{1}{2}\right]^\top$. Hence, the probability of measuring 000 is $|\langle 000|C|000\rangle|^2 = \frac{1}{4}$ and the probability of measuring 001 is $|\langle 001|C|000\rangle|^2 = 0$.

Note that, as in Sect. 2.1.1, we use a row vector for the measured computational basis state and a column vector for the initial state of the circuit.

Key message: an n-qubit state is transformed by $2^n \times 2^n$ matrices through matrix-vector multiplication. Measurement allows one to extract information from the state.

We will now take a first step towards using quantum circuits to our advantage. We focus on Boolean satisfiability (SAT), and will provide a naive approach to solving SAT using a quantum computer. Although this approach will not work, it will show how the *single* evaluation of the Boolean circuit on a quantum state will evaluate the Boolean function on all *exponentially-many* bitstrings as input.

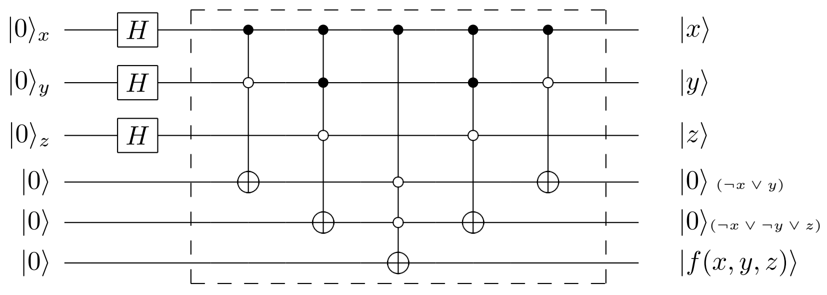
Consider the 3-CNF formula $f(x,y,z) = x \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y \vee z)$. We will use a reversible circuit to implement this function as a quantum circuit. We usually call such quantum circuit a *function oracle*. We want to compute f for every possible input $000, 001, \ldots, 111$. Therefore, we create a superposition of these states. This is done by applying a Hadamard gate to the qubits representing the input (see exercise below).

Now we will see what happens if we apply the function oracle f to a superposition of input states. Example 4 contains a simple example of a quantum circuit calculating $f = x \wedge y$. A more complicated example is the following one:

Example 6. Consider the 3-CNF formula

$$f(x,y,z) = x \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y \vee z). \tag{3}$$

The following circuit first applies Hadamard gates to get a superposition on the input qubits. Then it applies the function oracle for f (in the dashed rectangle).



The resulting state is $\sum_{x,y,z \in \{0,1\}^3} |x,y,z\rangle \otimes |f(x,y,z)\rangle$, or, written out in full,

$$\frac{1}{2\sqrt{2}}(|0000\rangle + |0010\rangle + |0100\rangle + |0110\rangle + |1000\rangle + |1010\rangle + |1100\rangle + |1111\rangle). \quad (4)$$

(Here, we omitted the two auxiliary qubits uncomputed to $|0\rangle$). Note that the single satisfying assignment (111) represents the solution to the satisfiability problem for f, because in Example 4, the only term which has a '1' at the fourth qubit, corresponding to $f(x,y,z)$, is $|1111\rangle$.

Exercise: Evaluate the circuit above in three steps. **(a)** Show that applying $H \otimes H \otimes H$ to the input $|000\rangle$ (the top three qubits in the circuit) yields a superposition over all 3-qubit computational-basis states. **(b)** Next, find the state when adding the register of the bottom three qubits (which is in the state $|000\rangle$) using the Kronecker product. **(c)** Lastly, using the Boolean function f as given in Eq. 3 (the circuit in the dashed rectangle above), verify that the resulting state indeed is Eq. 4.

We would like to extract the satisfying assignment '111' using measurement. Unfortunately, we see that measuring gives outcome 1111 with a probability of only $\left|\frac{1}{2\sqrt{2}}\right|^2 = \frac{1}{8}$. This probability is $\frac{1}{2^n}$ in general, where n is the number of variables to f. Thus, finding a satisfiable instance this way only succeeds with exponentially-small probability, much like the classical naive approach of randomly guessing bitstrings as input and evaluating f on them. $\diamond$

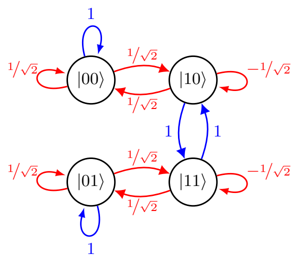
Boolean satisfiability & quantum computing. In Example 6 above, we saw a naive approach to solving Boolean satisfiability using quantum computing, which performed equally well as random guessing. However, using the famous quantum algorithm by Grover [15], the probability of measuring a satisfying assignment of f can be increased to arbitrarily high probability. Since every CNF formula f can be efficiently encoded as a reversible circuit, this solves the satisfiability problem for f. The Grover algorithm has runtime $O^*(\sqrt{2^n})$, where O^* omits polynomial factors. This is a quadratic speed-up compared to brute force and the best-known classical algorithm for k-SAT where k is unbounded.[a]

[a] Schöning's [32] and the PPSZ [26] algorithm deliver better guarantees for constant k, e.g., for 3-SAT, and can also be quantized [29].

2.3 Visualizing a Quantum Computation Using an Automaton

Armed with a well-defined notion of a quantum circuit, we can now continue the comparison with Markov chains in Sect. 2.1 and visualize an n-qubit quantum gate as an automaton: it has 2^n states, one for each computational-basis state, and we interpret the matrix of each gate U as the weighted adjacency matrix between these states. For example, consider 2 qubits, acted upon by the $CNOT$ gate and by the H gate on the first qubit:

$$H \otimes I = \begin{bmatrix} 1/\sqrt{2} & 0 & 1/\sqrt{2} & 0 \\ 0 & 1/\sqrt{2} & 0 & 1/\sqrt{2} \\ 1/\sqrt{2} & 0 & -1/\sqrt{2} & 0 \\ 0 & 1/\sqrt{2} & 0 & -1/\sqrt{2} \end{bmatrix}, \quad CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

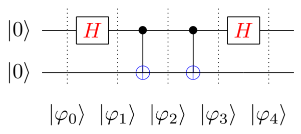


We will now use the following known result from graph theory:

> The entry in the product of adjacency matrices at row r and column c equals the sum of products of edge labels (a *path sum*) over all paths from node r to node c.

This somewhat complicated statement tells us that matrix multiplication can be visualized as path sums, a result we already implicitly used in the Markov chain case in Sect. 2.1.1 when observing that sequential composition of gates is represented by multiplication of transition matrices. To illustrate how this statement results in a visualization of quantum gates as paths in the automaton, we give the following example.

Example 7. Consider the circuit below; we will compute the entries in $(H \otimes I) \cdot CNOT \cdot CNOT \cdot (H \otimes I)$ corresponding to the transitions $|00\rangle \rightarrow |00\rangle$ and $|00\rangle \rightarrow |10\rangle$, first using matrix-vector multiplication, and then using paths in the automaton. For the former, it is straightforward to derive that $|\varphi_1\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$, hence $|\varphi_3\rangle = \frac{1}{\sqrt{2}} \cdot 1 \cdot 1 \cdot |00\rangle + \frac{1}{\sqrt{2}} \cdot 1 \cdot 1 \cdot |10\rangle$ (the second CNOT uncomputes the first). We use this to compute $|\varphi_4\rangle = (H \otimes I)|\varphi_3\rangle$ below.



$$\begin{aligned}
|\varphi_4\rangle &= \tfrac{1}{\sqrt{2}} \cdot 1 \cdot 1 \cdot \left(\tfrac{1}{\sqrt{2}}|00\rangle + \tfrac{1}{\sqrt{2}}|10\rangle + \tfrac{1}{\sqrt{2}}|00\rangle - \tfrac{1}{\sqrt{2}}|10\rangle\right) \\
&= \left(\tfrac{1}{\sqrt{2}} \cdot 1 \cdot 1 \cdot \tfrac{1}{\sqrt{2}} + \tfrac{1}{\sqrt{2}} \cdot 1 \cdot 1 \cdot \tfrac{1}{\sqrt{2}}\right)|00\rangle \\
&\qquad + \left(\tfrac{1}{\sqrt{2}} \cdot 1 \cdot 1 \cdot \tfrac{1}{\sqrt{2}} - \tfrac{1}{\sqrt{2}} \cdot 1 \cdot 1 \cdot \tfrac{1}{\sqrt{2}}\right)|10\rangle \qquad (5) \\
&= \left(\tfrac{1}{2} + \tfrac{1}{2}\right)|00\rangle + \left(\tfrac{1}{2} - \tfrac{1}{2}\right)|10\rangle \qquad (6)
\end{aligned}$$

Now the gate sequence $H - CNOT - CNOT - H$ means we should consider ‘red-blue-blue-red’ paths in the automaton. The factor $\frac{1}{2} + \frac{1}{2}$ in front of $|00\rangle$ in Eq. 6 is mirrored in the automaton by noting that there are two paths from $|00\rangle$ to itself: $|00\rangle \xrightarrow{1/\sqrt{2}} |10\rangle \xrightarrow{1} |11\rangle \xrightarrow{1} |10\rangle \xrightarrow{1/\sqrt{2}} |00\rangle$ and $|00\rangle \xrightarrow{1/\sqrt{2}} |00\rangle \xrightarrow{1} |00\rangle \xrightarrow{1} |00\rangle \xrightarrow{1/\sqrt{2}} |00\rangle$. These paths both have amplitude $1/\sqrt{2} \cdot 1 \cdot 1 \cdot 1/\sqrt{2} = 1/2$, and their sum $1/2 + 1/2$ is the amplitude of $|00\rangle$ in $|\varphi_4\rangle$ in Eq. 6 indeed (also note that the two products $1/2 \cdot 1 \cdot 1 \cdot 1/2$ are precisely the terms in front of $|00\rangle$ in Eq. 5). In contrast, the factor $1/2 - 1/2$ in front of $|10\rangle$ in Eq. 5 arises in the automaton as two paths from $|00\rangle$ to $|10\rangle$, one with amplitude $1/\sqrt{2} \cdot 1 \cdot 1 \cdot 1/\sqrt{2}$, and one with amplitude $1/\sqrt{2} \cdot 1 \cdot 1 \cdot -1/\sqrt{2}$, which are precisely the terms in front of $|10\rangle$ in Eq. 5. $\diamond$

Note that in the example above, the path sums have opposite sign, so they precisely cancel each other, implying that the transition $|00\rangle \to |10\rangle$ has amplitude zero! This cannot happen in a Markov chain as probabilities are never negative.

We thus see that the transition amplitude from state $|x\rangle$ to state $|y\rangle$, which we first found through linear algebra, can be found in the automaton by summing path contributions from $|x\rangle$ to $|y\rangle$, where each path contribution is the product of the edge labels of the path.

The automaton is useful because it shows a few properties of quantum circuits:

1. an **exponentially-sized state space** as function of number of qubits
2. the **combinatorial nature of the evolution of a quantum state through a circuit:** there can be **many paths** from one node to another, sometimes exponentially-many, and we need to track all of them to compute the output state's amplitudes
3. the many paths from one node to another will **interfere** as amplitudes, either **constructively** (the amplitudes amplify through addition, as in the example above for computing the amplitude of $|00\rangle$) or **destructively** (the amplitudes cancel, as for $|10\rangle$ above).

Item 1 and 2 also occur for probabilistic computation (see the Markov chain in Sect. 2.1) but item 3 is where quantum computing differs. Automated reasoning methods were often developed to solve scenarios where items 1 and 2 are present; we will see one approach in Sect. 3 that also illustrates item 3.

Key message: during a run of a quantum circuit, there can be exponentially many paths leading from one state to another. The amplitude contributions of these paths can constructively or destructively interfere.

2.4 Towards a Quantum Advantage

Application of a gate G once to a superposition of many computational-basis states $|b\rangle$ yields a sum of terms of the form $G|b\rangle$. This realization is particularly astounding in case we start out with an exponentially-large superposition, which can be created already with only linearly many gates in the number of qubits (recall the exercise above to compute $H^{\otimes n}|0\rangle^{\otimes n}$ for $n = 3$). However, using measurement, we can only read off a single bitstring from this superposition.

A plethora of quantum algorithms[8] [28] has been found whose quantum circuits provably need polynomially-fewer or exponentially-fewer gates than their classical counterpart.[9] Real-world quantum devices suffer from noise, and counteracting that noise might require additional resources that cancel the complexity-theoretic and real-world advantages of quantum computing. Arguably **the main open problem of quantum technologies is therefore to provide a real-world demonstration of a quantum advantage**. On the road to this goal, there are several tasks where formal methods could help: performance prediction of real-world devices through classical simulation of quantum circuits (sampling or computing the measurement distribution); optimizing circuits (e.g. fewer gates, circuit layouts following the topology of real-world chips, etc.); verification, specifically checking if two quantum circuits implement the same unitary matrix; finding a circuit that outputs a desired quantum state; transpilation to a gate set that real-world devices can natively run; etc.

In the remainder of this tutorial, we will focus on using weighted model counting or #SAT (Sect. 3) to perform classical simulation of quantum circuits.

[8] A quantum algorithm is a uniform family of quantum circuits (like in circuit complexity [2]).

[9] The exact statements depend on the used model (for example, whether the input is promised to be picked from a certain set). Quantum algorithms are typically complex, so we decided they are out of scope for this tutorial, where we aim to focus on analyzing quantum circuits using formal methods tools.

2.5 Our Scope: Quantum Circuit Simulation

In this tutorial, we will mainly consider the task of *classically simulating a quantum circuit* (that is, to design classical algorithms for this task). Formally, the task of simulating an n-qubit quantum circuit C is to find the probability of outcome $b \in \{0,1\}^n$ when the output state of C is measured, assuming that $|0\rangle^{\otimes n}$ is the input quantum state to C. Although simulation is #P-hard in general [23], so are many problems in formal methods, where solutions have been found that work well in practice. We should thus not be discouraged from tackling simulation and will see how to do so with #SAT in Sect. 3. In Sect. 4, we refer to extensions that tackle other important quantum circuit analysis tasks.

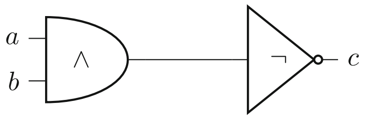
We focus here on *strong simulation* [7]: the problem of returning a probability for a certain computational basis state. In contrast, *weak simulation* asks to sample the probability distribution of measurement outcomes.

3 Reducing Quantum Computing to #SAT

Here we reduce quantum-circuit simulation to weighted model counting (weighted-#SAT). This section is partly based on [21], while the approach here effectively realizes the well-known path-sum approach [12].

3.1 SAT and #SAT

We denote $SAT(F) := \{\alpha \mid F(\alpha) = 1\}$ for the set of all satisfiable assignments of a propositional formula $F\colon \{0,1\}^V \to \{0,1\}$ over a finite set of Boolean variables V. We say that F is *satisfiable* if $SAT(F)$ is non-empty. We write an assignment α as a *cube* (a conjunction of literals, i.e., positive or negative variables), e.g., $a \wedge b$, or shorter ab.



The action of a classical circuit can be encoded by SAT constraints directly by representing each bit as a Boolean variable. For example, the Boolean constraint for the classical circuit C on the right is $F_C(V) = c \Leftrightarrow \neg(a \wedge b)$ over variables $V = \{a, b, c\}$. Given an input $a = 0$ and $b = 1$, the satisfying assignment is $\alpha = \overline{a}bc$, where $\alpha(c) = 1$ is the final state.

We denote $\#SAT(F) \triangleq |SAT(F)|$ for the *model count* of a formula F. A weight function $W\colon \{\overline{v}, v \mid v \in V\} \to \mathbb{R}$ assigns a real-valued weight to positive literals v (i.e., $v = 1$) and the negative literals $\overline{v}$ (i.e., $v = 0$). We say variable v is *unbiased* iff $W(v) = W(\overline{v}) = 1$. Given an assignment $\alpha \in \mathbb{B}^V$, let $W(\alpha(v)) = W(v := \alpha(v))$ for $v \in V$. For a propositional formula F over V and a weight function W, we define *weighted model counting* (#SAT) as:

$$\#SAT_W(F) \triangleq \sum_{\alpha \in SAT(F)} W(\alpha) \text{ where } W(\alpha) = \prod_{v \in V} W(\alpha(v)).$$

Example 8. The propositional formula $F = (v_1 \vee v_2) \wedge (\overline{v_1} \vee v_2) \wedge v_3$ over $V = \{v_1, v_2, v_3\}$ has two satisfying assignments: $\alpha_1 = v_1 v_2 v_3$ and $\alpha_2 = \overline{v_1} v_2 v_3$. We define the weight function W as $W(v_1) = -\frac{1}{2}$, $W(\overline{v_1}) = \frac{1}{3}$ and $W(v_2) = \frac{1}{4}$, $W(\overline{v_2}) = \frac{3}{4}$, while v_3 remains unbiased. The weight of F can be computed as $MC_W(F) = -\frac{1}{2} \times \frac{1}{4} \times 1 + \frac{1}{3} \times \frac{1}{4} \times 1 = -\frac{1}{24}$.

Why #SAT for tackling simulation? First, analysis tasks on quantum circuits are inherently functional problems, as the outcome often is a measurement probability (or amplitude). For the same reason, inference on Bayesian Networks [30,31] was done with weighted model counting. Here we use a similar approach by reducing the simulation of quantum circuits to weighted model counting.
In our #SAT encoding, we let each satisfying assignment encode a path in the automaton as discussed in Sect. 2.3. We then add weighted variables so that the weight for each assignment (representing a path) equals a part of the circuit's final amplitude. We also let the measurements constrain the encoding so that only the required paths remain. The weighted model counter ensures that the weights of all paths contributing to the measurement outcome are summed up (positive and negative).
Our encoding uses only linearly many variables and clauses in the number of gates plus the number of qubits. So, to encode a single gate, we require only a constant amount of clauses (around four), while encoding measurements require n (unit) clauses, one per qubit.

3.2 Encoding Quantum States Using (Weighted) Boolean Variables

Recall from Sect. 2 that a quantum state $|\phi\rangle$ is a specific linear combination of classical states, i.e.,

$$|\phi\rangle = \sum_{b \in \{0,1\}^n} \alpha_b |b\rangle.$$

Accordingly, we can simply reserve a single Boolean variable for every qubit and let the satisfying assignments of our formulae represent the state vector. Example 9 illustrates this. In what follows, we will write $F_{|\phi\rangle}$ for a similar encoding of $|\phi\rangle$.

Example 9. Let $|B\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |11\rangle)$. Let x be the variable for the first qubit and y be the variable for the second qubit. Written differently, we have: $|B\rangle = (\frac{1}{\sqrt{2}}|0\rangle_x - \frac{1}{\sqrt{2}}|1\rangle_x) \otimes |1\rangle_y$. The corresponding Boolean constraint is $F_{|B\rangle} = (x \vee \overline{x}) \wedge y$ where we assign the $W(x) = -\frac{1}{\sqrt{2}}$ and $W(\overline{x}) = \frac{1}{\sqrt{2}}$, leaving y unbiased. The (weighted) satisfying assignments are: $\{\overline{x}y \equiv \frac{1}{\sqrt{2}}|01\rangle, xy \equiv -\frac{1}{\sqrt{2}}|11\rangle\}$. $\diamond$

From now on, we will reserve the variables x, y, z for the first three qubits.

Exercise: Encode the so-called Bell state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ in weighted model counting.

3.3 Encoding Quantum Gates and Circuits to #SAT

Since the *NOT*, *CNOT* and *CCNOT* (Toffoli) gates are classical (reversible) gates, their encoding is easy. For instance, the Toffoli gate on input bits x, y, z and output bits x', y', z', only flips z, i.e., sets $z' \Leftrightarrow \neg z$, when both x and y are set to true, as explained in Sect. 2. Nothing changes in the quantum setting, except that we reserve the same variables now for the qubits. The Boolean encoding of these gates is thus as follows.

$$\begin{aligned} F_{NOT} \quad & (x, x') \triangleq x' \Leftrightarrow x \oplus 1 \quad = \quad x' \Leftrightarrow \neg x \\ F_{CNOT} \quad & (x, y, x', y') \triangleq y' \Leftrightarrow y \oplus x \wedge x' \Leftrightarrow x \\ F_{CCNOT} \quad & (x, y, z, x', y', z') \triangleq z' \Leftrightarrow z \oplus (y \wedge z) \wedge x' \Leftrightarrow x \wedge y' \Leftrightarrow y \end{aligned} \tag{7}$$

Observe that the above encoding determines the output variables x', y', z' in terms of the input variables, as illustrated in the following example.

Example 10. Let the input state be $|110\rangle$, where the corresponding Boolean constraint is $F_{|110\rangle} = x \wedge y \wedge \neg z$. After applying *CCNOT* gate, the output state is described by the constraint

$$F_{|110\rangle} \wedge F_{CCNOT} = (x \wedge y \wedge \neg z) \quad \wedge \quad (z' \Leftrightarrow z \oplus (y \wedge z) \wedge x' \Leftrightarrow x \wedge y' \Leftrightarrow y)$$

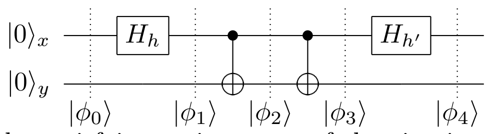
with satisfying assignment $x'y'z'$, which is interpreted as state $|111\rangle$.

In the (non-classical) case where we have superposition $\frac{1}{\sqrt{2}}(|011\rangle + |111\rangle)$, each basis state will be a satisfying assignment, e.g. yz with $W(x) = W(\overline{x}) = \frac{1}{\sqrt{2}}$, where the satisfying assignments are $\{xyz, \overline{x}yz\} \equiv \frac{1}{\sqrt{2}}(|011\rangle + |111\rangle)$. Applying a *CCNOT* gate ends up with two computational basis states (satisfying assignments) $\{\overline{x}yz, xy\overline{z}\} \equiv \frac{1}{\sqrt{2}}|011\rangle + |110\rangle$. $\diamond$

Recall that its gate semantics in Example 2. We will encode the gate again as a constraint $F_H(x, x', h)$, where x/x' is the qubit input/output variable and h is a separate variable representing the $\pm\frac{1}{\sqrt{2}}$ normalization. Notice in particular that the encoding should increase the number of satisfying assignments after introducing the x' variable. We achieve this by leaving x' unconstrained. The following encoding of the Hadamard gate also ensures that the negative weight only occurs for the case when $|1\rangle$ is the input and $|1\rangle$ is the output.

$$F_H(x, x', h) \quad \triangleq \quad h \iff (x \wedge x') \quad \text{with} \quad W(\overline{h}) = \frac{1}{\sqrt{2}} \quad W(h) = -\frac{1}{\sqrt{2}} \tag{8}$$

Example 11. The following circuit (identical to the one used in Sect. 2.3) computes the famous Bell state $|\phi_2\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ at time step 2, only to uncompute it again, ending up back in the $|00\rangle$ state. As will become apparent, this circuit nicely illustrates how the encoding handles constructive and destructive interference. In the circuit, we explicitly label the qubits with Boolean variables x, y and add a subscript to the Hadamard gates, to indicate that we will reserve an additional Boolean variable h or h' per gate.



We first show the satisfying assignments of the circuit encoding at each time step, where for the encoded Hadamard gates $F_H(x, x', h)$, we add an additional constraint $y \Leftrightarrow y'$ implementing the identity on the second qubit, i.e., $H \otimes I$.

$$\begin{aligned}
|\phi_0\rangle &\equiv F_{|\phi_0\rangle} = \neg x_0 \wedge \neg y_0 && \{\overline{x}_0\overline{y}_0\}\\
|\phi_1\rangle &\equiv F_{|\phi_1\rangle} = F_{|\phi_0\rangle} \wedge F_H(x_0, x_1, h) \wedge (y_0 \Leftrightarrow y_1) && \{\overline{h}x_1\overline{y}_1, \overline{h}\overline{x}_1\overline{y}_1\}\\
|\phi_2\rangle &\equiv F_{|\phi_2\rangle} = F_{|\phi_1\rangle} \wedge F_{CNOT}(x_1, y_1, x_2, y_2) && \{\overline{h}x_1\overline{y}_1x_2y_2, \overline{h}\overline{x}_1\overline{y}_1\overline{x}_2\overline{y}_2\}\\
|\phi_3\rangle &\equiv F_{|\phi_3\rangle} = F_{|\phi_2\rangle} \wedge F_{CNOT}(x_2, y_2, x_3, y_3) && \{\overline{h}x_1\overline{y}_1x_2y_2x_3\overline{y}_3, \overline{h}\overline{x}_1\overline{y}_1\overline{x}_2\overline{y}_2\overline{x}_3\overline{y}_3\}
\end{aligned}$$

It is worth noting that, in the final time step, the satisfying assignments of $|\phi_4\rangle \equiv F_{|\phi_4\rangle} = F_{|\phi_3\rangle} \wedge F_H(x_3, x_4, h') \wedge (y_3 \Leftrightarrow y_4)$ will be

$$\begin{aligned}
\{h'\overline{h}x_1\overline{y}_1x_2y_2x_3\overline{y}_3x_4\overline{y}_4 \equiv -\tfrac{1}{2}|10\rangle, \quad & \overline{h}'\overline{h}x_1\overline{y}_1x_2y_2x_3\overline{y}_3\overline{x}_4\overline{y}_4 \equiv \tfrac{1}{2}|00\rangle,\\
\overline{h}'\overline{h}\overline{x}_1\overline{y}_1\overline{x}_2\overline{y}_2\overline{x}_3\overline{y}_3x_4\overline{y}_4 \equiv \tfrac{1}{2}|10\rangle, \quad & \overline{h}'\overline{h}\overline{x}_1\overline{y}_1\overline{x}_2\overline{y}_2\overline{x}_3\overline{y}_3\overline{x}_4\overline{y}_4 \equiv \tfrac{1}{2}|00\rangle\},
\end{aligned}$$

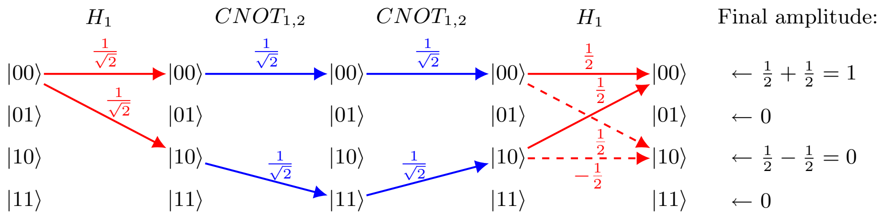
where we have $(\frac{1}{2} - \frac{1}{2})|10\rangle$ (destructive interference) and $(\frac{1}{2} + \frac{1}{2})|00\rangle$ (constructive interference). Recall in Sect. 2.3, we give the transition paths of states in the same circuit. Consider one of the paths from $|00\rangle$ to itself corresponding to the satisfying assignment $\overline{h}'\overline{h}\overline{x}_0\overline{y}_0x_1\overline{y}_1x_2y_2x_3\overline{y}_3\overline{x}_4\overline{y}_4$: $\underset{\overline{x}_0\overline{y}_0}{|00\rangle} \xrightarrow[\overline{h}]{1/\sqrt{2}} \underset{x_1\overline{y}_1}{|10\rangle} \xrightarrow{1} \underset{x_2y_2}{|11\rangle} \xrightarrow{1} \underset{x_3\overline{y}_3}{|10\rangle} \xrightarrow[\overline{h}']{1/\sqrt{2}} \underset{\overline{x}_4\overline{y}_4}{|00\rangle}$.

Here we omit the satisfying assignments for x_0 and y_0 in $SAT(F_{|\psi_t\rangle})$ with $t \in [1, 4]$ as each of them contains $\overline{x}_0\overline{y}_0$. We see that the satisfying assignments have a one-to-one mapping to the paths. $\diamond$

This encoding effectively realizes the well-known path sum approach [12] to the classical simulation of quantum circuits. It can be written in linear algebra as follows, where $U_0, U_1 \ldots, U_m$ are the (n-qubit) gates in the circuit and $|\boldsymbol{b}_1\rangle, |\boldsymbol{b}_2\rangle, \ldots, |\boldsymbol{b}_m\rangle$ are computational basis states.

$$U_m \cdots U_1 U_0 |00\ldots0\rangle = \sum_{\boldsymbol{b}_1,\boldsymbol{b}_2,\ldots,\boldsymbol{b}_m \in\{0,1\}^n} U_m \cdot |\boldsymbol{b}_m\rangle\langle\boldsymbol{b}_m| \ \ldots \ |\boldsymbol{b}_2\rangle\langle\boldsymbol{b}_2| \cdot U_1 \cdot |\boldsymbol{b}_1\rangle\langle\boldsymbol{b}_1| \cdot U_0 \cdot |00\ldots0\rangle$$

The previous example shows that the #SAT encoding does not "merge" paths ending in the same computational basis state, as illustrated below (dashed edges cancel each other out).



Key message: Given a universal quantum circuit, consisting of Toffoli and Hadamard gates, we can construct a Boolean formula whose satisfying assignments represent the circuit's output quantum state.

3.4 Encoding Measurements

Recall in Sect. 2.2.3, measuring all qubits in computational basis obtains the probability of an outcome $|b\rangle$ for $b \in \{0,1\}^n$. In a circuit with gates $U_1, \ldots, U_m$ and an input state $|\varphi_0\rangle$, the output state $|\varphi_m\rangle = U_m \cdots U_1|\varphi_0\rangle$ can be decomposed into basis states as $|\varphi_m\rangle = \sum_{b\in\{0,1\}^n} \alpha_b|b\rangle$, where the amplitudes can be computed by weighted model counting $\#SAT_W(F_{|\varphi_m\rangle} \wedge F_{|b\rangle}) = \alpha_b$, where $F_{|\varphi_m\rangle} = F_{U_m} \wedge \cdots \wedge F_{U_2} \wedge F_{U_1} \wedge F_{|\varphi_0\rangle}$ follows the encoding in Sect. 3.3 and $F_{|b\rangle}$ is the encoding of basis state $|b\rangle$ as shown in Sect. 3.2. The probability then equals $(\alpha_b)^2$.

Example 12. Reconsider Example 11 with measurements on all qubits of the output state. To get the probability of measuring basis state $|b\rangle = |00\rangle$, the constraint $F_{|b\rangle} = \overline{x}_4 \wedge \overline{y}_4$. should be conjoined to the final state: $F_{|\varphi_4\rangle} \wedge F_{|b\rangle}$. We then have $SAT(F_{|\varphi_4\rangle} \wedge F_{|b\rangle}) = \{\overline{h}'\overline{h}x_1\overline{y}_1\overline{x}_2\overline{y}_2x_3\overline{y}_3\overline{x}_4\overline{y}_4, \overline{h}'\overline{h}\overline{x}_1\overline{y}_1\overline{x}_2\overline{y}_2\overline{x}_3\overline{y}_3\overline{x}_4\overline{y}_4\}$. The amplitude of $|00\rangle$ is $MC_W(F_{|\varphi_4\rangle} \wedge F_{|b\rangle}) = W(h)W(h') + W(h)W(h') = 1$, thus the resulting probability is $1^2 = 1$. $\diamond$

Exercise: In the above example, compute the probability of measuring with basis state $|01\rangle$.

By adding the measurement constraint, only the "paths" to the basis state we measure would be left. In the sense of satisfying assignments, as shown in the previous example, we will keep the satisfying assignments with variables on the final time step encoding the measured basis state ($\overline{h'}\overline{h}x_1\overline{y}_1\overline{x}_2\overline{y}_2x_3\overline{y}_3\overline{x}_4\overline{y}_4 \equiv \frac{1}{2}|00\rangle, \overline{h'}\overline{h}\overline{x}_1\overline{y}_1\overline{x}_2\overline{y}_2\overline{x}_3\overline{y}_3\overline{x}_4\overline{y}_4 \equiv \frac{1}{2}|00\rangle$) and discard others ($h'\overline{h}x_1\overline{y}_1x_2y_2x_3\overline{y}_3x_4\overline{y}_4 \equiv -\frac{1}{2}|10\rangle$, $\overline{h'}\overline{h}\overline{x}_1\overline{y}_1\overline{x}_2\overline{y}_2\overline{x}_3\overline{y}_3x_4\overline{y}_4 \equiv \frac{1}{2}|10\rangle$).

4 Wrap-Up

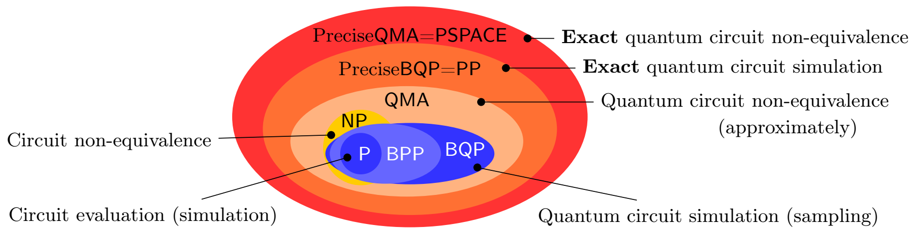
This tutorial detailed how the basic task of *classical simulation* of quantum computing can be done using #SAT, which illustrates the inherent combinatorial nature of quantum computing as well as interference, which amplifies or cancels amplitudes of the output quantum state. We now explain some important open problems in quantum computing and demonstrate, based on related work, how similar methods can solve them.

Current quantum computers will have limited numbers of qubits that are noisy [27]. For this reason, investment in the field only took off after the invention of quantum error correction [14], which realizes the ideal quantum circuit model on noisy hardware. However, the added complexity of error correction requires additional resources since the logical error-corrected qubits consist of many physical qubits. So to attain a quantum advantage earlier, we need to optimize quantum algorithms to use few qubits and respect the gate sets and topology (physical connectivity of qubits) of the many different quantum hardware types.

Therefore, we need to optimize quantum circuits, synthesize them in different gate sets and verify their correctness. The latter starts with checking whether an optimized circuit implements the same operations as its original (equivalence checking), but also involves checking quantum Hoare logic [41]. Finally, error-corrected quantum computers will work in tandem with classical computers to monitor the process [13], which requires solving tasks strongly related to the ones discussed above.

Formal methods can indeed be extended to solve the above tasks. For instance, decision diagrams have been used to check the equivalence of quantum circuits [6, 38] and model counting [22] as well. For a more detailed overview of the successes of formal methods in this domain, we refer readers to a historical overview [36] (in which all authors of this tutorial were involved). As an honorable mention, we note that there are also approaches based on diagrammatic reasoning using the ZX calculus [8, 40], which has also been used for circuit optimization [18], on planning [33] and on abstract interpretation [4].

However, the applications of our methods can be viewed in an even broader context. The features that make quantum compilation difficult are shared with the computationally-hard aspects of many problems in quantum physics and quantum chemistry, such as computing the energy of the ground state of a physical system [17] or simulating a many-body system [25]. Indeed, many of these problems in physics are in the quantum analogs of the complexity classes NP and P, i.e., in QMA [5] and BQP [19]; see the figure below for how these quantum complexity classes relate to classical ones (reproduced from [10]). Therefore, any progress in tackling hard problems for quantum circuits can be directly used to solve the very problems that quantum physicists and chemists struggle with on a daily basis (using the default reductions between BQP- and QMA-complete problems). This is important since even if the ideal error-corrected quantum computer never gets built, we are still stuck with the myriad of "quantum-hard" problems that nature poses—the very reason why Feynman proposed building a quantum computer by inverting the problem in the first place (see Sect. 1).


PreciseQMA=PSPACE
Exact quantum circuit non-equivalence
PreciseBQP=PP
Exact quantum circuit simulation
QMA
Quantum circuit non-equivalence (approximately)
NP
Circuit non-equivalence
P
BPP
BQP
Circuit evaluation (simulation)
Quantum circuit simulation (sampling)

We finish by giving some references for further reading: Lipton and Regan's accessible introduction to quantum computing for computer scientists [20], a seminal textbook by Nielsen and Chuang [24], the lecture notes of Ronald de Wolf [9] and Watrous' detailed monograph on quantum information [39].

Acknowledgements. This publication is part of the project Divide & Quantum (with project number 1389.20.241) of the research program NWA-ORC which is (partly) financed by the Dutch Research Council (NWO). This work was supported by the Dutch National Growth Fund, as part of the Quantum Delta NL program. The third author acknowledges the support received through the NWO Quantum Technology program (project number NGF.1582.22.035).

References

1. Aharonov, D.: A simple proof that Toffoli and Hadamard are quantum universal. arXiv preprint quant-ph/0301040 (2003)
2. Arora, S., Barak, B.: Computational Complexity: A Modern Approach. Cambridge University Press, Cambridge (2009). https://doi.org/10.1017/cbo9780511804090

3. Arute, F., et al.: Quantum supremacy using a programmable superconducting processor. Nature **574**, 505–510 (2019). https://www.nature.com/articles/s41586-019-1666-5
4. Bichsel, B., Paradis, A., Baader, M., Vechev, M.: Abstraqt: analysis of quantum circuits via abstract stabilizer simulation. Quantum **7**, 1185 (2023). https://doi.org/10.22331/q-2023-11-20-1185, http://dx.doi.org/10.22331/q-2023-11-20-1185
5. Bookatz, A.D.: QMA-complete problems. arXiv:1212.6312 (2012)
6. Burgholzer, L., Wille, R.: Advanced equivalence checking for quantum circuits. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **40**(9), 1810–1824 (2020). https://doi.org/10.1109/tcad.2020.3032630
7. Chen, Y., Chen, Y., Kumar, R., Patro, S., Speelman, F.: Qseth strikes again: finer quantum lower bounds for lattice problem, strong simulation, hitting set problem, and more. arXiv preprint arXiv:2309.16431 (2023)
8. Coecke, B., Duncan, R.: Interacting quantum observables: categorical algebra and diagrammatics. New J. Phys. **13**(4), 043016 (2011). https://doi.org/10.1088/1367-2630/13/4/043016
9. De Wolf, R.: Quantum computing: lecture notes. arXiv preprint arXiv:1907.09415 (2019)
10. Deshpande, A., Gorshkov, A.V., Fefferman, B.: Importance of the spectral gap in estimating ground-state energies. PRX Quant. **3**(4), 040327 (2022)
11. Feynman, R.P.: Simulating physics with computers. Int. J. Theor. Phys. **21**(6), 467–488 (1982). https://doi.org/10.1007/BF02650179
12. Feynman, R., Hibbs, A., Styer, D.: Quantum Mechanics and Path Integrals. Dover Books on Physics, Dover Publications (2010). https://books.google.nl/books?id=JkMuDAAAQBAJ
13. Fowler, A.G., Mariantoni, M., Martinis, J.M., Cleland, A.N.: Surface codes: towards practical large-scale quantum computation. Phys. Rev. A **86**(3), 032324 (2012)
14. Gottesman, D.: Stabilizer codes and quantum error correction. Ph.D. thesis, California Institute of Technology (1997)
15. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, pp. 212–219 (1996)
16. Kemeny, J.G., Snell, J.L., et al.: Finite Markov Chains, vol. 26. van Nostrand, Princeton (1969)
17. Kempe, J., Kitaev, A., Regev, O.: The complexity of the local Hamiltonian problem. SIAM J. Comput. **35**(5), 1070–1097 (2006)
18. Kissinger, A., van de Wetering, J.: Reducing the number of non-Clifford gates in quantum circuits. Phys. Rev. A **102**, 022406 (2020). https://doi.org/10.1103/PhysRevA.102.022406, https://link.aps.org/doi/10.1103/PhysRevA.102.022406
19. Kitaev, A.Y., Shen, A., Vyalyi, M.N.: Classical and quantum computation. Am. Math. Soc. (2002)
20. Lipton, R.J., Regan, K.W.: Introduction to Quantum Algorithms via Linear Algebra. MIT Press, Cambridge (2021)
21. Mei, J., Bonsangue, M., Laarman, A.: Simulating quantum circuits by model counting. In: CAV 2024, (accepted for publication). Springer (2024). Pre-print available at arXiv:2403.07197)
22. Mei, J., Coopmans, T., Bonsangue, M., Laarman, A.: Equivalence checking of quantum circuits by model counting. In: IJCAR (accepted for publication) (2024). Pre-print available at arXiv:2403.18813)

23. den Nest, M.V.: Classical simulation of quantum computation, the Gottesman-Knill theorem, and slightly beyond. arXiv:0811.0898 (2008)
24. Nielsen, M.A., Chuang, I.L.: Quantum Information and Quantum Computation, vol. 2, no. 8, p. 23. Cambridge University Press, Cambridge (2000)
25. Orús, R.: A practical introduction to tensor networks: matrix product states and projected entangled pair states. Ann. Phys. **349**, 117–158 (2014). https://doi.org/10.1016/j.aop.2014.06.013, https://www.sciencedirect.com/science/article/pii/S0003491614001596
26. Paturi, R., Pudlák, P., Saks, M.E., Zane, F.: An improved exponential-time algorithm for k-sat. J. ACM (JACM) **52**(3), 337–364 (2005)
27. Preskill, J.: Quantum computing in the NISQ era and beyond. Quantum **2**, 79 (2018)
28. Quantum algorithm zoo. https://quantumalgorithmzoo.org/. Accessed 25 Apr 2024
29. Rennela, M., Brand, S., Laarman, A., Dunjko, V.: Hybrid divide-and-conquer approach for tree search algorithms. Quantum **7**, 959 (2023). https://doi.org/10.22331/q-2023-03-23-959
30. Roth, D.: On the hardness of approximate reasoning. Artif. Intell. **82**(1–2), 273–302 (1996)
31. Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: AAAI, vol. 5, pp. 475–481 (2005)
32. Schoning, T.: A probabilistic algorithm for k-sat and constraint satisfaction problems. In: 40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039), pp. 410–414. IEEE (1999)
33. Shaik, I., van de Pol, J.: Optimal layout synthesis for quantum circuits as classical planning. arXiv:2304.12014 (2023)
34. Shi, Y.: Both Toffoli and controlled-NOT need little help to do universal quantum computing. Quant. Inf. Comput. **3**(1), 84–92 (2003)
35. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science, pp. 124–134. IEEE (1994)
36. Thanos, D., et al.: Automated reasoning in quantum circuit compilation. In: Preproceedings of SPIN2024 (2024). https://spin-web.github.io/SPIN2024/assets/preproceedings/SPIN2024-paper6.pdf
37. Toffoli, T.: Reversible computing. In: de Bakker, J., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 632–644. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10003-2_104
38. Viamontes, G.F., Markov, I.L., Hayes, J.P.: Checking equivalence of quantum circuits and states. In: 2007 IEEE/ACM International Conference on Computer-Aided Design, pp. 69–74 (2007). https://doi.org/10.1109/ICCAD.2007.4397246
39. Watrous, J.: The Theory of Quantum Information. Cambridge University Press, Cambridge (2018)
40. van de Wetering, J.: Zx-calculus for the working quantum computer scientist. arXiv:2012.13966 (2020)
41. Ying, M.: Floyd-Hoare logic for quantum programs. ACM Trans. Program. Lang. Syst. (TOPLAS) **33**(6), 1–49 (2012)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

No Risk, No Fun
A Tutorial on Risk Management

Mariëlle Stoelinga[1,2](✉)

[1] Formal Methods and Tools, University of Twente, Enschede, The Netherlands
m.i.a.stoelinga@utwente.nl
[2] Department of Software Science, Radboud University, Nijmegen, The Netherlands

Abstract. The aim of this tutorial is to explain to the formal methods community the area of risk management and its most prominent concepts: the definition of risk, strategies for managing risk, the risk management cycle, and the role of ISO standards.

For each of these concepts, I explain how formal methods relate and contribute, making risk management more accountable: systematic, transparent, and quantitative. I will also argue that viewing Formal Methods through the lens of risk management, and making the relevance of formal methods in risk analysis explicit, helps our community to better communicate the merits of formal methods to industry.

Keywords: Risk management · formal methods · uncertainty

1 Introduction

1.1 Formal Methods and Risk Management

Risk Management. Risk management [4,26] is something we do every day: we lock our houses to prevent burglary; our health insurance covers the financial consequences of hospital visits; cars are checked yearly to prevent failures; we back up data to not lose valuable information; we wear seat belts when driving; we double check if we have not forgotten our phones, etc.

In industry, such decisions are made at large: access policies determine which employees can enter the building; companies insure their employees against work accidents; regular maintenance keeps production plants up and running; back ups are performed to not lose valuable data; helmets and safety glasses protect employees against injuries; the four eyes principle – reviewing critical tasks by at least two people – enhances accuracy. However, such measures cost time and money and are also inconvenient —insurance and COVID masks are prime

This work was partially funded by the NWO grants NWA.1160.18.238 (PrimaVera), and KICH1.ST02.21.003 (ZORRO), the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 101008233, and the ERC Consolidator Grant 864075 (*CAESAR*).

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 447–468, 2025.
https://doi.org/10.1007/978-3-031-71177-0_26

examples here. Thus, a key concern is to select effective measures to lower the most prominent risks [26]: *The overall goal of risk management is to support decision-making on (cost-)effective measures that keep risks below an acceptable level.*

Formal Methods. Formal methods refer to mathematically rigorous techniques for the specification, development, analysis, and verification of software and hardware systems [15,22]. In this tutorial, I adopt a broader definition, following e.g. [55]: Rather than focusing on software and hardware, I will consider any kind of system. These include physical systems, such as biological and financial systems, but also services, procedures and missions. The broader definition, also taken in [23,55], enables a better comparison with Risk Management, which also covers many domains, such as technological, environmental, financial, and social risks. Moreover, Formal Methods have actually been applied to a wide variety of systems, including biological systems [39,52], chemical systems [1], business processes [41], and human behavior.

What sets formal methods apart from other disciplines is not the systems considered, but rather the methodological approach: modeling languages to conveniently specify the system under study, as well as languages to model their properties; formal syntax and semantics for these languages; rigorous analysis techniques that have been proven correct; compositional approaches to build large systems from smaller components, where especially the interaction between components matters. However, even with a more narrow scope of formal methods, there are strong links with risk management.

1.2 Formal Methods Versus Risk Management

Formal methods and risk management have strong links. They strive for the same goals, namely high-quality and reliable systems without surprises. Their means are quite different, though: Formal methods focus on mathematical methods, while risk management uses informal methods.

Formal Methods for Risk Management. The area of Formal Methods has made numerous relevant contributions to the area of risk management. These fall into three categories.

First, the field of Formal Methods has developed and strengthened a wide variety of risk assessment frameworks. Risk assessment is part of the risk management cycle (cf. Sect. 5) concerned with the identification, prioritization, and evaluation of risks. The area of Formal Methods has equipped a broad range of industrial risk assessment frameworks (such as fault trees [18], reliability block diagrams [43], the AADL language) with rigorous semantics, more modeling power and efficient analysis algorithms. These techniques enabled analyses of systems that were not possible before [12,32].

Second, these techniques were possible due to more fundamental advances in the underlying stochastic analysis methods, enabling the analysis of systems with gigantic state spaces for a plethora of properties and metrics. Relevant approaches include Bayesian analysis [46], scenario optimization [13], Monte Carlo simulation [53], stochastic and statistical model checking [37], reinforcement learning [56], stochastic optimization and control [11]. Since uncertainty is a key ingredient of risk, stochastic and statistical analysis is a relevant area for risk analysis.

Finally, numerous methods have been developed to alleviate risks in software systems: rigorous specification, design and software verification, model-based testing, algorithms and architectures for fault-tolerant computing, monitoring and run-time verification, and debugging are all relevant techniques to reduce system risk. The risk management terminology classifies these as *risk treatment strategies*.

Many of these contributions were published in venues where formal methods intersect with (technical) subfields of risk analysis, such as dependability analysis, reliability engineering, and safety analysis, with relevant conferences, such as DSN, FMICS, FM, NFM, QEST and SAFECOMP.

Risk Management for Formal Methods. Despite these relevant contributions, risk thinking is not in the standard repertoire of researchers in formal methods. This is a pity for three reasons: The first argument I like to put forward is that *The area of Formal Methods is in a unique position to make risk management more accountable: more systematic, transparent, and quantitative.* It is very well known, especially through the work of Nobel laureate Daniel Kahneman [33,34], that people have bad intuitions for risk. His numerous experiments repeatedly show how poorly people can assess chances and risks. Numerous cognitive biases have been identified that influence people's perception of risk. According to Kahneman, there are two systems at work in our brains: System I, which makes decisions quickly and automatically, and System II, which is slow, systematic, and rational. System I is handy when we need to locate objects or interpret facial expressions, but it is not suitable for assessing probabilities. That is better left to System II. Even though there is no scientific evidence, it is my firm believe that, through their rigorous approach, Formal Methods foster System II thinking.

My second argument is that *viewing Formal Methods through the lens of risk management provides a better perspective of where Formal Methods matter most.* In particular, by addressing the most prominent risks, What do people and organizations care about most? What are their concerns? Which formal methods are appropriate to address these concerns? Where in the system development cycle can formal methods contribute most, and which methods are appropriate? This point of view aligns well with the pledge in the recent Manifesto for Applicable Formal Methods [24] that advocates integrative practices.

Finally, incorporating formal methods into risk management can amplify their importance and influence. Whereas few people know formal methods, almost everybody knows risk management. Framing formal methods in terms of risk management can increase the relevance and contributions of our field to the general public, politicians, and business managers.

To reap the benefits above, a comprehension of risk management is needed —exactly the goal of this tutorial.

1.3 This Tutorial

Objective. The objective of this tutorial is to provide an overview of the basic concepts, principles, and terminology of risk management for researchers in Formal Methods. Explicating the strong links between the fields of Formal Methods and Risk Management allows formal methods researchers to better align their research with concerns and practices in industry and society.

With focus on terminology and concepts, this tutorial is neither very technical nor very formal.

Didactic Set Up. This tutorial starts by setting the basic concepts then zooms out on the organizational processes supporting risk management.

Thus, I first discuss the various definitions of risk and related concepts, and delve into the three main elements of risk: *objectives*, *consequences*, and *uncertainty*. Once the main system risks have been identified, the question of what actions are to be taken arises: which interventions (if any) are needed to decrease the system risks? Four common risk intervention strategies will be discussed: *tolerate*, *terminate*, *transfer*, and *treat*. Next, the tutorial will cover the risk management cycle (PDCA), the starting point for most organizations. It will also discuss the formal methods recommended by ISO standards.

Organization. Section 2 discusses the definitions of risk and related concepts. Section 3 reviews the elements of risk: objectives, consequences, and uncertainty, and Sect. 4 discusses the four risk treatment strategies: tolerate, terminate transfer, and treat. The PDCA cycle and the role standards are discussed in Sects. 5 and 6 respectively. Section 7 concludes the proposal.

2 Risk and Risk Management

2.1 What Is Risk?

A first problem encountered in risk management, perhaps especially for the Formal Methods community, is that there is no standard definition of risk. Not only has the concept of risk evolved over time, just today many different definitions of risk exist across different disciplines, professional societies, scientists, and international standardization bodies. Here are some of them:

- The Institute of Risk Management [28] defines risk as the combination of the probability of an event and its consequences.
- The Threat Analysis Group [59] defines risk as asset, threat, and vulnerability.

- The PRINCE2 method [7] defines risk as: An uncertain event or set of events that, should it occur, will have an effect on the achievement of objectives.
- Fenton & Neil [21] assume risks to be unfavorable events influenced by factors. Such factors and their interactions might be random or uncertain.
- Kaplan & Garrick [35] define risk as a set of triplets (s_i, p_i, x_i) of a scenario s_i, a probability p_i and a consequence level x_i.
- The ISO 31000:2009 standard [30] on risk management defines risk as: the effect of uncertainty on objectives.

Sources of debate are (1) whether risk include both negative and positive consequences, or only negative ones and (2) whether the likelihood of events should be interpreted as a probability, or in the broader terms of uncertainty. Despite significant efforts, the Society for Risk Analysis has concluded that settling for a single definition is not realistic.

A summary of various definitions of risk over time can be found in [48] and in more technical terms in [3]. In his seminal work *Against the Gods: The Remarkable Story of Risk*, the economist and financial historian Peter Bernstein provides an account of the history of risk from ancient Greece until today [10].

In this tutorial, I will use the ISO 31000 definition of risk:

the effect of uncertainty on objectives.

This definition is relatively widely accepted and stresses the emphasis on goals. For example, the U.S. National Institute of Standards and Technology (NIST) has also adopted this definition in the context of cybersecurity.

Remarks. Some remarks on the definitions of risk: (1) All definitions are application-independent. They apply to financial risks for a venture capitalist, high-tech risks concerning self-driving cars, and everyday risks like choosing travel insurance. (2) These definitions apply to various artifacts-products, processes, services, and missions-collectively termed *systems*. Whether it is a medical device, a banking procedure, or a military mission, the same risk management principles apply. (3) The definition is relevant to all phases of a system's life cycle: requirement gathering, design, implementation, operation, and dismantling.

2.2 Risk Categories

Risks are often classified based on the organizational level they address:

- *Strategic risks* concern the organization's mission and long-term strategy, concerning e.g., market expansion, technological innovation, and mergers.
- *Tactical risks* affect the implementation of strategies in operations, projects, and processes, e.g., supply chain disruptions or regulatory changes.
- *Operational risk* are specific to the organization's internal processes and include human errors, technology failures, or fraud incidents.
- *Compliance risks* involve adherence to law, regulations, and internal policies.

Most research in formal methods focuses on operational and compliance risks, analyzing risk in airplanes, self-driving cars, and robots. Formal methods to address tactical risks have also been developed, especially in the context of enterprise risk modeling frameworks and business process modeling.

2.3 Related Terminology

Risk Versus Risk Level. Some older definitions, define risk as the statistically expected loss, i.e., the product of probability and impact [49]. This definition reduces risk to quantitative aspects; it is better to refer to this product as the risk level of an event e:

$$\mathsf{RiskLevel}(e) = \mathsf{Prob}(e) \times \mathsf{Impact}(e)$$

While standard, the definition of risk level does have some limitations: It does not take into account the evolution of failure probabilities over time, and neither the fact that the probability and impact of an event are uncertain themselves. Nevertheless, the risk level is popular to get a quick overview of the importance of several events in terms of risk.

Risk Versus Hazard. A hazard is defined as a source of danger [6]. Risk includes the likelihood that the hazard will lead to actual loss, injury, or damage.

Risk Versus Resilience. A popular term in the context of risk is resilience, i.e., the ability to withstand, adapt to, and recover from disruptive events or crises. Thus, resilience can be seen as a risk treatment strategy (cf. Sect. 4) that emphasizes impact reduction. While risk focuses on negative outcomes, resilience emphasizes the ability to recover and thrive despite challenges.

3 The Ingredients of Risk

This section reviews the three main ingredients of risk from the ISO 31000 definition: *objectives*, *effect* (described as impact) and *uncertainty*. First, I introduce a common visualization tool: the risk matrix.

3.0 The Risk Matrix

A risk matrix, a.k.a. risk priority heat map, is a popular tool to visualize risks. As shown in Fig. 1, this diagram plots the likelihood of an event against the severity (or impact). Such maps yield a quick overview of the risk landscape and help prioritize risks, as critical risks require the most attention and low risks the least.

Despite their merits, risk matrices should be handled with care. Several phenomena are not taken into account with sufficient consideration:

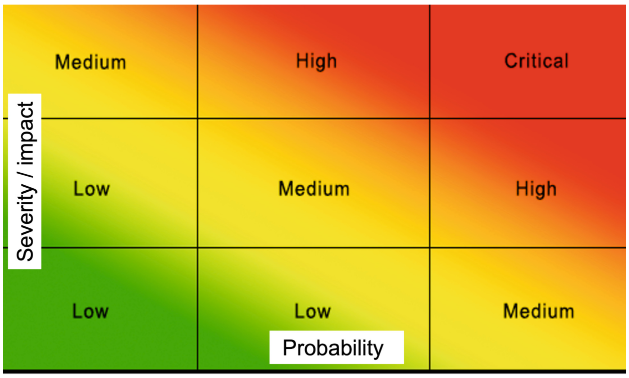


Fig. 1. Risk matrix

- The risk matrix presents a snapshot of the probability and likelihood of events at a certain point in time. Their evolution over time is not reflected in the matrix.
- Often, the probability and likelihood of an event are uncertain themselves. This could be accommodated in the risk matrix by plotting the events not as a dot but rather as an area, but this is not often done.
- Dependencies and causal relations between risks cannot be accommodated.

Moreover, special attention should be paid to so-called *HILP* events (High Impact, Low Probability). Examples are nuclear power plant explosions and the Ever Given cargo ship that blocked the Suez canal in 2021, disrupting supply chains worldwide. HILP events sit in the very top corner of the risk matrix and are categorized as medium risk level. However, due to their enormous impact, HILP events require special attention.

Example. Figure 2 shows a risk matrix from the Global Risks Report 2022 by the World Economic Forum [61]. It visualizes the largest risks perceived by companies worldwide.

3.1 Ingredient 1: Objectives

Let us turn to the ingredients of risk. Recall the ISO 31000 definition of risk as the effect of uncertainty on objectives. In fact, one can say that, according to this definition, there is no risk, if there is no objective. Although such objectives are usually not specified in a formal way, they should be formulated as accurately as possible. Three guidelines are important:

- Objectives should be formulated in a SMART (Specific, Measurable, Attainable, Realistic and Time-bound) way, where concreteness and realism are the most important focal points.
- The objectives should be shared and agreed on with all stakeholders.
- Objectives should not stimulate perverse behavior.

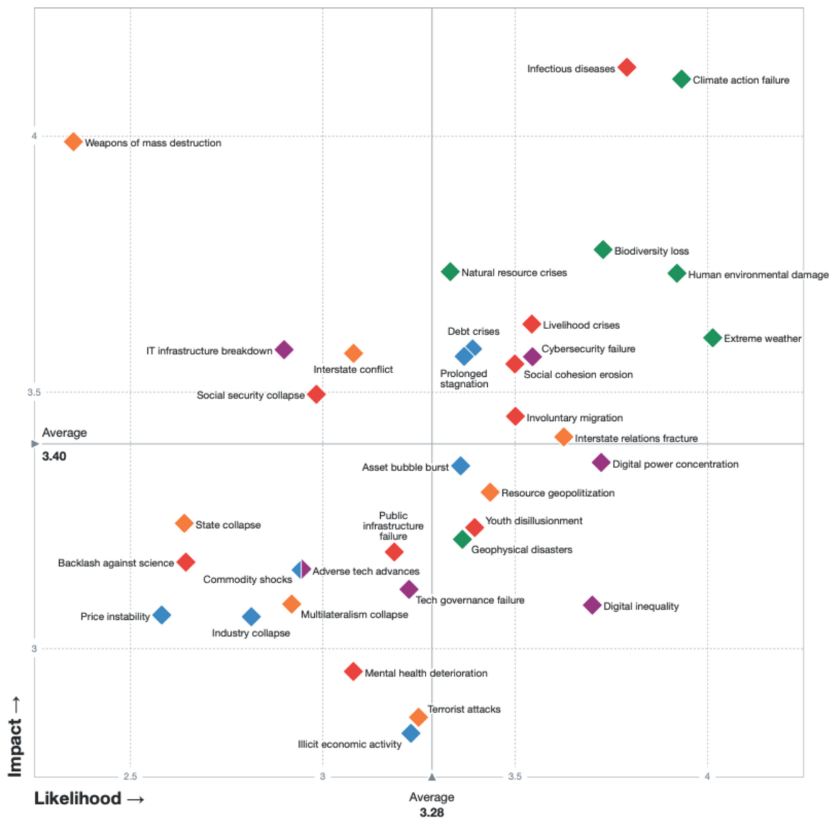


Fig. 2. Risk matrix from World Economic Forum

3.2 Ingredient 2: Impact

The second ingredient is impact. Evidently, different events can lead to a variety of outcomes. Typically, these outcomes are classified into five major impact classes:

- *Cost.* Accidents are expensive, including covering the costs to recover from injuries and cyber incidents. Budget overruns in (software) projects are also common.
- *Time.* Sometimes, risk causes delays. Natural disasters can stop the supply chain and delay project delivery. In software projects, delays are a recurring risk.
- *Reputation.* Reputation is often underestimated, but is often considered one of the most severe consequences, having a long-term impact on operations and relationships with stakeholders.
- *Health & safety.* These risks encompass harm and loss of life as a result of motor accidents, fires, explosions, errors in health care and natural disasters.

Technology-related safety risks include exposure to toxic chemicals, defective airbags, faulty medical devices, and privacy breaches.
– *Quality.* Compromises in the quality of a product or service are serious, as they often directly affect the company's mission. In software systems, bugs are notorious, pertaining to the pervasive view that software testing should be risk-based.

Recently, a sixth factor came up, namely *sustainability*: the impact on environmental and social needs for future generations.

Severity Classes. In practice, standard severity classes are often used, rating the impact on a point scale. The following 4-point scale is used by hospitals for patient safety, showing how the medical field operationalizes severity. This scale can be seen as a discretization of severity levels combined with a description specific to the setting.

Scale	Patient safety
1. Minor	Discomfort
2. Moderate	Light injury
3. Critical	Permanent injury
4. Catastrophic	Patient death

Quantitative Formal Methods. Many formal methods quantify impacts as real numbers, in the form of rewards, cost, price, or utility, for example, in reinforcement learning techniques, stochastic games, Markov reward models, timed priced games etc.

3.3 Ingredient 3: Uncertainty

Uncertainty implies a situation in which a person does not have the necessary information to precisely describe, prescribe, or predict an event or its characteristics. Uncertainty comes in two flavors [16,42]:

– *Aleatoric uncertainty* originates from the word *alea*, Latin for dice, and refers to uncertainty stemming from natural fluctuations.
– *Epistemic uncertainty* originates from the Greek word *epistéme*, which means knowledge. Epistemic uncertainty comes from our lack of knowledge.

Aleatoric Uncertainty. Traditionally, the mathematical analysis of risk has focused on aleatoric uncertainty, using the laws of probability theory. As Bernstein states [10]:

> Probability theory is an instrument for organizing, interpreting and applying information. As one genius idea was piled on top of another, quantitative techniques of risk management have helped trigger the ideas of modern times. [..] Without the command of probability theory and other instruments of risk management, engineers could never have designed the great bridges that span our rivers, [..] polio would still be maiming children, no airplanes would fly, and space travel would just be a dream.

Numerous formal methods have been developed to handle aleatoric risks: These can be divided into two approaches: first, formal methods have made enormous contributions to the analysis of stochastic models. These include classic models, such as Markov chains, Markov decision processes, and Bayesian networks. These achievements have enabled better analysis of existing risk models, such as fault trees, reliability block diagrams, and AADL. Stochastic risk models are especially popular in the areas of probabilistic safety analysis (PSA) and reliability engineering.

Epistemic Undertainty. Epistemic uncertainty refers to uncertainty arising from a lack of knowledge or understanding about the system risks. It stems from limitations in available data, models, and parameters.

Epistemic uncertainty can be reduced through further research, data collection, or refinement of models and theories. However, it may never be fully eliminated because of inherent limitations in human understanding or the complexity of the system being studied. Properly addressing epistemic uncertainty is crucial to effectively mitigate risks.

The Combination. Several approaches exist that combine aleatoric and epistemic uncertainty. These especially include methods in which the probabilistic parameters are subject to uncertainty.

Prominent models include Bayesian belief methods [21], where practitioners can update their beliefs about both the parameters of a model and the variability in the data as new information becomes available. Other methods include methods based on fuzzy probability theory [17,62], interval Markov chains [8], parametric Markov models [9], hidden and partially observable Markov models [54]. All these models are based on stochastic methods —aleatoric uncertainty— but allow for uncertainty in the parameters of the model.

Uncertainty is a large concept, with many different angles and interpretations, see e.g., [5,38] for an interpretation in safety analysis.

3.4 Uncertainty: Black Swans and the Rumsfeld Matrix

When it comes to epistemic uncertainty, two concepts have emerged in the field: Black swans and the uncertainty matrix by Rumsfeld.

Black Swans. Important in the context of uncertainty is the concept of so-called Black Swans. Black swans were coined by Taleb [58] and are a metaphor for high-profile, hard-to-predict, and rare events with significant impacts that often catch people by surprise. Examples are the 2008 financial crises and the COVID pandamic.

Black swans have challenged traditional risk assessment methods, including, perhaps especially formal risk models based on aleatoric/stochastic analysis, because they were unable to foresee these black swans. Although black swans are difficult to anticipate, effective risk management strategies should include means to mitigate these, e.g. via robust contingency plans, resilience-building measures, and adaptive frameworks to mitigate their potential impacts.

The Rumsfeld Matrix of Uncertainty. The Rumsfeld matrix categorizes information and knowledge based on their levels of certainty and awareness. It originates from former U.S. Secretary of Defense Donald Rumsfeld and uses four categories for comprehending and addressing uncertainties:

1. *Known Knowns* are the well-understood risks, including known hazards, historical data, and established patterns. They can be effectively managed via established risk assessment and mitigation strategies.
2. *Known Unknowns* represent risks that are recognized but not fully understood. The key to managing these risks is to obtain better information via research, analysis, and exchange of information, as well as thorough risk analyses and scenario planning.
3. *Unknown Knowns* refers to risks that exist but are not consciously recognized. These may include hidden vulnerabilities, cultural biases, and blind spots. Effective risk management requires again better information.
4. *Unknown Unknowns.* are the risks beyond current awareness, i.e., the black swans. As the greatest challenge to risk management, these require an agile approach, using resilience and contingency plans.

Clearly, it is very difficult to mitigate all black swans, e.g., how to prepare for a next pandemic that can be completely different from the last one? As phrased in a quote often attributed to Niels Bohr, one of the fathers of quantum mechanics [60]: *Prediction is difficult, especially when it is about the future* (Fig. 3).

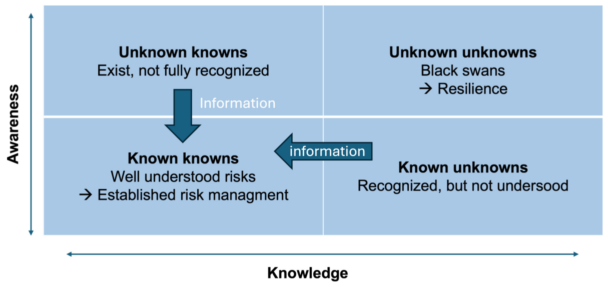


Fig. 3. The Rumsfeld matrix

4 Risk Strategies

After identifying system risks and classifying them according to likelihood and probability, the question is what to do with these risks. Four ways of dealing with risk exists, called *risk strategies.*

Tolerate. This strategy entails accepting risks as they are, taking no additional action—no risk, no fun. Plenty of examples exist, since we all drive, fly, walk. Products are released without being fully tested or verified.

Terminate. This is the opposite of tolerate: Stop, or do not start, an activity that is perceived as too risky. The decision of the aviation authorities in 2019 to not allow the Boeing 737 max to fly passengers is an example of the risk termination strategy.

Transfer. At times, risk can be transferred to other parties. Insurance serves as a prime example, in which the monetary hazards associated with theft, fire, or medical treatments are covered by an insurance plan. Other examples include outsourcing, where an entire task, including its intrinsic risks, is delegated to a third-party entity. However, it is impossible to completely transfer risks: insurance policies may handle the financial implications of medical care, but do not alleviate the other effects. Furthermore, in the case of outsourcing, there is always the danger that the entity to which the task was delegated fails to perform.

Treat. Risk treatment is an important strategy, as it finds mitigation measures to reduce potential risks. Mitigation measures are also called *controls.* There are two types:

(a) *Impact reduction* reduces the effect of hazards after they occur by taking corrective measures. Examples are safety devices (helmets, seat belts, air bags), monitoring systems (smoke detectors), and fail-safe mechanisms (which return the system to a safe state after an incident happened, for example, the emergency lane offering a refuge after a car accident). Impact reduction measures from software engineering include run-time verification and exception handling.

(b) *Likelihood reduction* implements preventive measures that reduce the likelihood that an event will occur. Examples are training of personnel (driving licenses are a typical example), regular maintenance schedules of machinery, regular software updates, and strict security protocols to prevent unauthorized information access. Many practices in software engineering, either formal or informal, fall into this category: rigorous specification, verification, validation, testing, etc.

4.1 The Application of Risk Strategies

It is important to re-assess risks after measures have been divised. Have all prominent risks been addressed? Have new risks been introduced? Risks that remain after measures have been taken are called *residual risks.*

It is good practice to implement a combination of preventive and corrective measures for important risks. Preventive measures have the advantage that the incident does not occur at all, so no damage is done. However, risk can often not be completely predicted, and therefore corrective measures are useful. This is especially the vision of resilience engineering: predicing risks is difficult, and black swans can always occur, therefore adaptability and flexibility are of utmost importance.

4.2 Risk Management Versus Dependability Engineering

Risk management is closely related to dependability engineering. Dependability [6] is defined as: *the ability to deliver service that can justifiably be trusted*, and refined into *the ability to avoid service failures that are more frequent and more severe than is acceptable.* Terminology of risk and dependability are closely related.

In their seminal paper, Avizienis et al. [6] break down the dependability landscape into attributes that reflect dependability concerns, threats that endanger dependability, and means to improve dependability, see Fig. 4.

Definitions. When considering the service as the primary objective of a system, it becomes clear that the concept of dependability is linked to (absence of) risk.

Attributes as Impact Classes. Six dependability attributes are identified: *availability* meaning readiness for correct service; *reliability* meaning continuity of correct service; *safety* meaning absence of catastrophic consequences on the users and environment; *integrity* meaning the absence of improper system alterations; *maintainability* meaning ability to undergo modifications. These can be considered as refinement of the risk impact classes from Sect. 3.2: With a focus on technical correctness, availability, reliability, confidentiality, integrity, and maintainability refine the quality class. The safety attribute immediately corresponds to the safety impact class.

Means as Strategies. Finally, dependability means can be viewed as risk strategies discussed in Sect. 4. Fault prevention means to prevent the occurrence or introduction of faults, and is therefore a preventive risk reduction strategy. Fault tolerance means to avoid service failures in the presence of faults and thus is a corrective risk reduction strategy. Fault removal means to reduce the number and severity of faults: again a preventive risk reduction strategy. Fault forecasting means to estimate the present number, the future incidence, and the likely consequences of faults, and is a risk assessment activity.

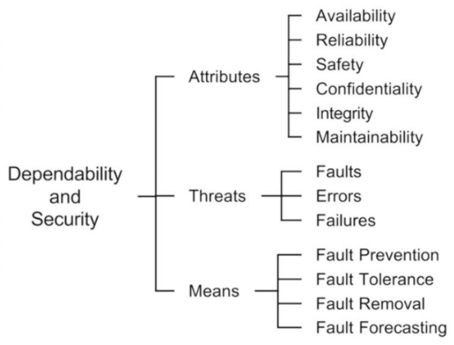


Fig. 4. Taxonomy for dependability attributes, threats and means

Actually, many formal methods can be viewed as risk strategies/ dependability means: Formal requirements specification, verification, validation can all be seen as fault prevention means, and thus as preventive measures. Debugging is a fault removal technique. Run-time verification is, when combined with e.g. fail-safe mechanisms, a mean for fault tolerance. Code metrics are fault forecasting means.

5 Risk Management

Risk management refers to coordinated activities to direct and control an organization with respect to risk [26]. Virtually all organizations manage their risks through the Plan-Do-Check-Act (PDCA) cycle, and many use its concretization in the ISO 31000 standard. The latter provides concrete steps to select appropriate risk strategies, deciding how risks should be treated and which interventions are appropriate.

This section covers both the PDCA cycle and the ISO 31000 standard, as well as the role of formal methods.

5.1 The Risk Management Cycle: Plan-Do-Check-Act

The Plan-Do-Check-Act (PDCA) cycle, also known as the Demming cycle, is a systematic process for continuous improvement of processes and products [57]. As illustrated in Fig. 5, this cycle proceeds in four steps. Specialized to risk management, these are as follows.

1. *Plan*: Establish goals, identify and assess risks, develop mitigation strategies.
2. *Do*: Implement risk management strategies and allocate necessary resources.
3. *Check*: Monitor the effectiveness of the strategies, and report findings.
4. *Act*: Improve and update plans to ensure continuous risk management.

Little information is available on the relation between the PDCA cycle and the use of formal methods. However, since the PDCA cycle is designed for any improvement process, it is also applicable to formal verification activities: Set the goals of the verification, perform the verification, check if the verification yields the desired results, and update plans to improve both the system under verification and the verification process itself.

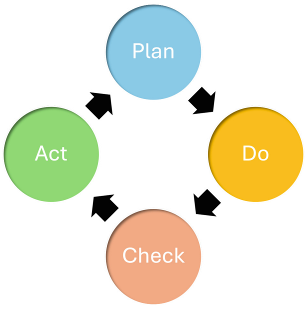


Fig. 5. PDCA cycle

5.2 The Risk Management Process

Several frameworks exist that refine and concretize the PDCA cycle. The ISO 31000 is a family of generic standards, applicable in many contexts. Other risk management frameworks, such as COSO [44], are more specialized for enterprise risks. ISO 31000 [30] provides principles, vocabulary and a process for any organization to assess and treat risks. Formal methods are especially useful during the Risk Assessment phase.
As illustrated in Fig. 6, the process consists of several steps:

1. *Establish the context*, and especially the goals.
2. *Identify risks,* mapping the risks that threaten the goal.
3. *Analyze the risks* finding the root causes and factors that contribute to the risks.
4. *Evaluate risks* according to their likelihood and impact.
5. *Treat risks* finding effective measures.

5.3 Formal and Informal Methods for Risk Assessment

Performing a proper risk analysis is not easy and requires domain knowledge. For example, it is not trivial to identify all relevant risks in a self-driving car or nuclear plant.

There are several risk frameworks to support the risk assessment process. These frameworks offer a systematic procedure to identify risks in different classes, find root causes, and help determine their impact. The level of formality varies from very informal to very formal.

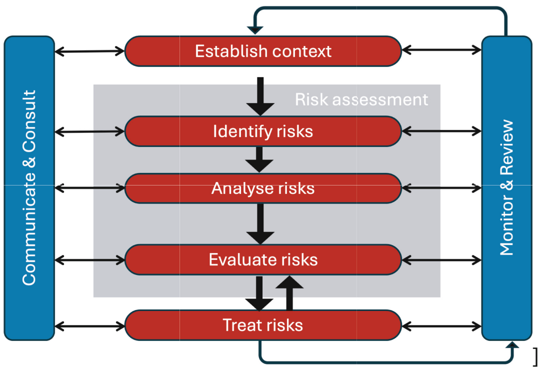


Fig. 6. Steps in the ISO31000 standard for risk management

Text-Based Methods. Textual approaches provide systematic methods for exploring components or behaviors in complex systems and list all findings in textual form or a table. Common approaches are failure mode effect analysis (FMEA) [50], and Hazard & operability studies (HAZOP) [36].

Architectural Methods. These methods take an architectural system model as a starting point, decomposing a system into a number of interacting components, annotating these with potential risks. Such architectural methods are especially common for systems with large software components, but can be used in any domain with complex system designs. Some prominent examples include the Architectural Analysis & Design Language AADL [20], the AltaRica framework [2,47], the Safety Analysis Modeling Language (SAML) [25].

Domain Specific Methods. These methods have been specifically developed for risk analysis. These include fault tree analysis [18,51], reliability block diagrams [43], event trees [19], and bowtie diagrams [14]. All of these methods provide visual means to capture system behavior and offer different analysis possibilities, for example, (stochastic) model checking, Monte Carlo simulation, or dedicated computation methods.

Various organizations dealing with safety-critical systems, including NASA, ESA, the nuclear industry, and the US Federal Aviation Administration, have recognized that a single analytical approach is usually insufficient for effective risk management. Consequently, they suggest a combination of approaches.

Finally, is important to realize that risk models, like many other models in computer science, do not formulate an objective truth, like in Newtonian mechanics. Rather, these models serve decision making and reflect the best information currently available. Moreover, in my experience, creating risk models at design time can lead to design improvements that prevent risks from happening all together: *the journey is the destination.*

6 ISO Standards, Risk Management and Formal Methods

6.1 The Role of ISO Standards in Risk Management

The International Standardization organization ISO is an independent, non-governmental organization that develops voluntary international standards for quality, safety, and efficiency in products, services, and systems.

These standards cover a wide range of industries and technologies, from manufacturing and technology to food safety and healthcare. ISO standards typically require organizations to establish systematic approaches to quality management, information security, health and safety, and more, by setting appropriate policies, procedures, and processes to monitor the outcomes. Several standards recommend the use of formal methods.

If companies and organizations meet the criteria for a certain standard, they can obtain certification for that standard. Such an accreditation is advantageous, as it bolsters an organization's credibility and trust among customers. In addition, regulatory bodies or governments may require adherence to specific ISO standards as part of legal or contractual obligations.

Some standards are developed with other organizations such as IEEE or IEC, as reflected in their name. Names may also include the year of publication, reflecting the specific version.

6.2 ISO Standards for Software Systems

Some noteworthy standards related to software systems are the following.

ISO/IEC TR 5469: Artificial Intelligence—Functional Safety and AI Systems. This standard outlines the role of AI in safety-related systems, classes, and compliance levels. It provides guidance regarding the specification, design, and verification of functionally safe AI systems, or how to apply AI technology for functions that have safety-related effects. *ISO/IEC 42001: AI management systems* encompasses the Plan-Do-Check-Act cycle for AI management systems.

ISO/IEC/IEEE 90003:2018 Software Engineering. This standard is part of the ISO 9000 family on quality management. The 90003 standard provides guidelines for the acquisition, supply, development, operation, and maintenance of software and support services.

ISO/IEC/IEEE 12207:2018 Systems and Software Engineering Software Life Cycle Processes. Whereas ISO 90003 relates to software purchase, the 12207 standard relates to software development, setting requirements for the software life cycle process: agreement, organizational, technical management, and technical processes. The latter includes business or mission analysis, stakeholder needs, requirements, architecture, design, implementation, integration, verification, validation, operation, maintenance, and disposal.

6.3 ISO Standards Recommending Formal Methods

Several ISO standards, mostly related to safety-critical systems, recommend formal methods during design and verification. Here are some notable instances:

ISO 26262: Road Vehicles—Functional Safety. [29] concerns the functional safety of electrical and electronic systems in road vehicles for the entire automotive safety lifecycle: management, development, production, operation, and decommissioning. Automotive Safety Integrity Levels (ASIL) set risk levels, based on the probability and consequences of safety hazards.

ISO 22163:2023 Railway Applications, Railway Quality Management System [31]. It refines ISO 9001:Quality management systems with specific requirements for application in the railway sector.

IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems offers guidelines for implementing, designing, deploying, and maintaining safety-related systems, achieved through a safety life cycle and a probabilistic failure assessment.

A key concept in dependability analysis in safety-critical system is the notion of *Safety Integrity Level (SIL)*, referred to as automotive SIL (ASIL) in the automotive industry [27]. (A)SIL levels range from 1 (low) to 4 (highest). For hardware, the device must meet strict limits on failure probability for (a rigorously defined notion of) dangerous failure.

Based on the (A)SIL level, the use of formal methods is recommended. ISO 50128 recommends formal methods for SIL 1 and 2, and highly recommends them for SIL 3 and 4. Interestingly, ISO 26262 recommends formal methods, but highly recommends semi-formal methods, such as UML and SySML.

6.4 Research on Formal Methods for ISO Compliance

Establishing a safety analysis in the context of ISO standards can be challenging. Various formal validation and verification techniques have applied to several ISO standards, such as ISO 6262 [40]. One significant hurdle is the requirement that tools used for developing safety-critical systems must be certified. An overview of approaches to demonstrate compliance with ISO standards, which can serve as a foundation for further application of formal methods, is provided in [45].

7 Conclusion

This tutorial provides an overview of risk management concepts, principles and techniques and their relation to formal methods.

Formal methods are in a good position to stimulate System II thinking. In this way, they set a good basis for accountable risk making: *systematic*, so that no risk are overlooked; *transparent*, since models explicate the information that risk decisions are based on; and *quantitative*, based on facts rather than on feelings.

References

1. Lano, K., Bicarregui, J., Kan, P.: Experiences of using formal methods for chemical process control specification. Control. Eng. Pract. **8**(1), 71–79 (2000)
2. Arnold, A., Griffault, A., Point, G., Rauzy, A.: The AltaRica formalism for describing concurrent systems. Fundam. Inf. **40**, 109–124 (2000)
3. Aven, T.: The risk concept-historical and recent development trends. Reliab. Eng. Syst. Saf. **99**, 33–44 (2012)
4. Aven, T.: The reliability science: Its foundation and link to risk science and other sciences. Reliab. Eng. Syst. Saf. **215**, 107863 (2021)
5. Aven, T., Reniers, G.: How to define and interpret a probability in a risk and safety setting. Saf. Sci. **51**, 223–231 (2013)
6. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. **1**, 11–33 (2004)
7. AXELOS: Prince2 6th edition glossary of terms (2016). https://www.axelos.com/resource-hub/glossary/prince2-6th-edition-glossaries-of-terms
8. Bacci, G., Delahaye, B., Larsen, K.G., Mariegaard, A.: Quantitative analysis of interval markov chains. In: Olderog, E.-R., Steffen, B., Yi, W. (eds.) Model Checking, Synthesis, and Learning. LNCS, vol. 13030, pp. 57–77. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-91384-7_4
9. Badings, T.S., Jansen, N., Junges, S., Stoelinga, M., Volk, M.: Sampling-based verification of CTMCs with uncertain rates. In: Shoham, S., Vizel, Y. (eds.) Proceedings of the 34th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 13372, pp. 26–47. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13188-2_2
10. Bernstein, P.L.: Against the Gods: The Remarkable Story of Risk. Wiley (1998)
11. Bertsekas, D.P.: Dynamic Programming and Optimal Control, vol. I, 4th Edition. Athena Scientific (2005)
12. Bozzano, M., Cimatti, A., Katoen, J., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability and performance analysis of extended AADL models. Comput. J. **54**(5), 754–775 (2011)
13. Campi, M.C., Carè, A., Garatti, S.: The scenario approach: a tool at the service of data-driven decision making. Annu. Rev. Control. **52**, 1–17 (2021)
14. Center for Chemical Process Safety: Bow Ties in Risk Management. Wiley (2018)
15. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. ACM Comput. Surv. (CSUR) **28**(4), 626–643 (1996)
16. Der Kiureghian, A., Ditlevsen, O.: Aleatory or epistemic? Does it matter? Struct. Saf. **31**(2), 105–112 (2009)
17. Dong, W.M., Shah, H., Wongt, F.: Fuzzy computations in risk and decision analysis. Civ. Eng. Syst. **2**(4), 201–208 (1985)
18. Dugan, J.B., Bavuso, S.J., Boyd, M.A.: Fault trees and sequence dependencies. In: Annual Reliability and Maintainability Symposium, pp. 286–293 (1990)
19. Ericson, C.A.: Event tree analysis. In: Hazard Analysis Techniques for System Safety, pp. 223–234. WILEY (2005)
20. Feiler, P.H., Gluch, D.P., Hudak, J.: The architecture analysis & design language (AADL): an introduction (2006)
21. Fenton, N., Neil, M.: Risk Assessment and Decision Analysis with Bayesian Networks. CRC Press (2011)

22. Garavel, H., Beek, M.H., Pol, J.: The 2020 expert survey on formal methods. In: ter Beek, M.H., Ničković, D. (eds.) FMICS 2020. LNCS, vol. 12327, pp. 3–69. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58298-2_1
23. Gibbins, P.: Chapter 13 - what are formal methods? In: Ince, D., Andrews, D. (eds.) The Software Life Cycle, pp. 278–290. Butterworth-Heinemann (1990)
24. Gleirscher, M., van de Pol, J., Woodcock, J.: A manifesto for applicable formal methods. Softw. Syst. Model. **22**(6), 1737–1749 (2023)
25. Güdemann, M., Ortmeier, F.: A framework for qualitative and quantitative model-based safety analysis. In: 2010 IEEE 12th International Symposium on High Assurance Systems Engineering (2010)
26. Hopkin, P.: Fundamentals of Risk Management: Understanding, Evaluating and Implementing Effective Risk Management. Kogan Page, 5th edn. (2018)
27. Houtermans, M.: SIL and Functional Safety in a Nutshell, 2nd edn. Prime Intelligence (2014)
28. Institute of Risk Management: IRM's risk management standard (2002). https://www.theirm.org/what-we-do/what-is-enterprise-risk-management/irms-risk-management-standard/
29. International Organization for Standardization: ISO 26262: Road vehicles - functional safety. ISO Standard (2018). https://www.iso.org/standard/26262.html
30. International Organization for Standardization: ISO 31000: Risk management – guidelines. ISO Standard (2018). https://www.iso.org/standard/65694.html
31. International Organization for Standardization: ISO22163: Railway applications; railway quality management system. ISO Standard (2023). https://www.iso.org/standard/22193.html
32. Junges, S., Guck, D., Katoen, J., Rensink, A., Stoelinga, M.: Fault trees on a diet: automated reduction by graph rewriting. Formal Aspects Comput. **29**(4), 651–703 (2017)
33. Kahneman, D.: A perspective on judgment and choice: mapping bounded rationality. Am. Psychol. **58**(9), 697–720 (2003)
34. Kahneman, D.: Thinking, Fast and Slow. Farrar, Straus and Giroux (2011)
35. Kaplan, S., Garrick, B.J.: On the quantitative definition of risk. Risk Anal. **1**(1), 11–27 (1981)
36. Kletz, T.: Hazop and Hazan: Identifying and Assessing Process Industry Hazards, 4th edn. Institution of Chemical Engineers (1999)
37. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_11
38. Lindley, D.V.: Understanding Uncertainty. Wiley (2006)
39. Lück, A., Wolf, V.: A stochastic automata network description for spatial DNA-methylation models. In: Hermanns, H. (ed.) MMB 2020. LNCS, vol. 12040, pp. 54–64. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43024-5_4
40. Makartetskiy, D., et al.: (User-friendly) formal requirements verification in the context of ISO26262. Eng. Sci. Technol. Int. J. **23**(3), 494–506 (2020)
41. Mannel, L.L., van der Aalst, W.M.P.: Discovering process models with long-term dependencies while providing guarantees and filtering infrequent behavior patterns. Fundam. Informaticae **190**(2–4), 109–158 (2024)
42. Matthies, H.G.: Quantifying uncertainty: modern computational representation of probability and applications. In: Extreme Man-Made and Natural Hazards in Dynamics of Structures, pp. 105–135. NATO Security through Science Series (2007)
43. Modarres, M., Kaminskiy, M.P., Krivtsov, V.: System reliability analysis. In: Reliability Engineering and Risk Analysis: A Practical Guide. CRC Press (2016)

44. Moeller, R.R.: COSO Enterprise Risk Management: Establishing Effective Governance, Risk, and Compliance Processes. Wiley (2011)
45. Myklebust, T., Stålhane, T.: Functional Safety and Proof of Compliance. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86152-0
46. Pearl, J.: Causality: Models, Reasoning, and Inference. Cambridge University Press (2000)
47. Point, G., Rauzy, A.: AltaRica: constraint automata as a description language. J. Européendes Systémes Automatisés **33**, 1033–1052 (2006)
48. Rasborg, K.: Ulrich Beck. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-89201-2
49. Rasmussen, N.: An assessment of accident risks in U.S. commercial nuclear power plants. Tech. rep., US Nuclear Regulatory Commission (1975)
50. Rausand, M., Barros, A., Hoylan, A.: Qualitative system reliability analysis. In: System Reliability Theory. Models, Statistical Methods, and Applications. Wiley (2020)
51. Ruijters, E., Stoelinga, M.: Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools. Comput. Sci. Rev. **15–16**, 29–62 (2015)
52. Schivo, S., et al.: Modeling biological pathway dynamics with timed automata. IEEE J. Biomed. Health Inf. **18**(3), 832–839 (2014)
53. Shonkwiler, R.W., Mendivil, F.: Explorations in Monte Carlo Methods. Springer, Cham (2009). https://doi.org/10.1007/978-3-031-55964-8
54. Spaan, M.T.J.: partially observable markov decision processes. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning, vol. 12, pp. 387–414. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-27645-3_12
55. Staunstrup, J.: Formal design methods. In: A Formal Approach to Hardware Design, pp. 1–12. Springer US (1994). https://doi.org/10.1007/978-1-4615-2764-0
56. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (1997)
57. Tague, N.R.: Plan–do–study–act cycle. In: The Quality Toolbox, pp. 390–392. ASQ Quality Press (2005)
58. Taleb, N.N.: The Black Swan: The Impact of the Highly Improbable. Random House (2007)
59. Threat Analysis Group: Threat, vulnerability, and risk: commonly mixed-up terms (2010). https://www.threatanalysis.com/2010/05/03/threat-vulnerability-risk-commonly-mixed-up-terms/
60. Wikiquote contributors: Niels bohr (2024). https://en.wikiquote.org/wiki/Niels_Bohr. Accessed 09 Jun 2024
61. World Economic Forum: The Global Risks Report, 13th Edition (2022)
62. Zadeh, L.A.: Fuzzy sets. Inf. Control **8**(3), 338–353 (1965)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Runtime Verification in Real-Time with the Copilot Language: A Tutorial

Ivan Perez[1(✉)], Alwyn E. Goodloe[2], and Frank Dedden[3]

[1] KBR @ NASA Ames Research Center, Houston, USA
ivan.perezdominguez@nasa.gov

[2] NASA Langley Research Center, Hampton, USA
a.goodloe@nasa.gov

[3] System F Computing, Rotterdam, The Netherlands
frank@systemf.dev

Abstract. Ultra-critical systems require high-level assurance, which cannot always be guaranteed at compile time. The use of runtime verification (RV) enables monitoring of these systems during runtime, to detect illegal states early and limit their potential consequences. This paper is a tutorial on RV using Copilot, an open-source runtime verification framework actively used by NASA to carry out experiments with robots and unmanned aerial vehicles. Copilot monitors are written in a compositional, stream-based language, which the framework automatically translates into real-time C code that satisfies static memory requirements suitable to run on embedded hardware. Copilot includes multiple libraries that extend the core functionality with higher-level constructs, Boyer-Moore majority voting, and a variety of Temporal Logics (TL), resulting in robust, high-level specifications that are easier to understand than their traditional counterparts.

1 Introduction

Embedded systems are used in a wide range of applications, ranging from televisions and cellphones to automobiles, aircraft and ships. In all of these applications, we want the system to function correctly, but those systems that are *safety critical*, where failure can result in injury or death of a human, warrant special attention [14]. To achieve the necessary level of reliability, both hardware and software of safety-critical systems need to be of very high quality. Formal verification techniques are one method for achieving the level of reliability required in safety-critical systems. Generally, formal verification is based on mathematically proving correctness properties of a model of the system under study. Although there have been considerable advances in industrial-scale formal methods, they remain too expensive to apply them on most projects.

Runtime verification (RV) [2,10,11] is a verification technique that has the potential to enable the safe operation of safety-critical systems that are too complex to formally verify or fully test. In RV, the system is monitored *during execution*, to detect and respond to property violations that take place during operation. When an unsafe state is detected, the monitor invokes system-specific

© The Author(s) 2025

routines to report or recover from the violation. RV detects when properties are violated during *during runtime*, so it does not constitute a proof of correctness, but is a significant improvement over testing alone.

RV concerns itself with the detection of faults and off-nominal conditions. Upon detection of a property violation, the system under observation (SUO) could switch to a backup system, transfer control to a pilot or operator on the ground, degrade performance, log the error for posterior analysis, or take some other corrective action. The specifics of how faults are handled are mission-specific, and not the subject of RV itself.

Correct implementation of these monitors and the RV subsystem is crucial for the safe operation of the complete system and the success of the mission as a whole. The introduction of errors in the RV subsystem could disable or affect other subsystems, or lead to suboptimal deviations from a mission. In resource-constrained environments and time-critical systems, runtime monitors are commonly implemented in C due to performance and memory constraints. This results in low-level code that is error-prone, hard to understand and difficult to maintain.

In this paper we present Copilot, a runtime verification framework to write high-level specifications. Copilot is implemented as a stream-based, deeply embedded domain-specific language in Haskell. Streams are used to specify monitors, which denote functions that detect when properties are violated. Once a monitor is triggered, a user-defined function is called to take appropriate action. Our framework provides a constrained set of operations to define and combine streams, which guarantees that they are well-formed. The language also relies on dependent types, to enable safe use of non-primitive data structures, like structs and arrays. Copilot translates definitions into a MISRA-compliant subset of C99. MISRA C is a set of guidelines for developing C code targeting real-time embedded systems, which promotes code safety and security. For instance, the guidelines constrain the design with predictable memory requirements and real-time guarantees. To run the monitor on an embedded system, the generated C99 code can be compiled for a target platform and integrated with the system under observation. Additional Copilot libraries extend the core language with higher-level constructs, and temporal logic [15,20].

This paper is structured as follows: Sect. 2 provides a brief history of the Copilot framework. Section 3 is an introduction to the trace theoretic view of RV, sampling, and RV instrumentation. Section 4 has a "hello world" style motivating example. Section 5 introduces the Copilot specification language which simplifies prior versions of the language [17,19] and extends it with notions of arrays and structs. Section 6 demonstrates how to specify runtime monitors using basic stream-level functions as well as different temporal logics. In Sect. 7, we discuss how to integrate Copilot monitors into the larger system being monitored.

2 History

The Copilot project began in September 2008 when NASA awarded Galois, Inc. and the National Institute of Aerospace a contract ("Monitor Synthesis for Software Health Management"[1]) to perform research in the area of runtime verification applied to hard real-time distributed systems with a concentration on avionics. Focusing on hard real-time embedded systems imposed the following constraints: monitors should not affect the system under observation (SUO) in a way that changes the functionality of the system, requires re-certification, interferes with timing, or exceeds size, weight, and power (SWAP) constraints. The four most significant design decisions made in the early days were: that the RV framework that came to be called Copilot would be implemented as a Haskell-embedded domain-specific language (EDSL), that Copilot would favor lightweight instrumentation and thus employ sampling as a means to observe the executing system, that runtime monitors must run in constant space and constant time, and the fourth significant design decision was that the specification language would be a stream-based data-flow language inspired by Lustre [4] and LOLA [6]. Note that Lustre has an existing user base in the aerospace industry, meaning that stream-based languages have proven acceptable in that domain.

The first Copilot prototypes focused on evolving the specification language and constructing an interpreter. During this time period, several open-source tools were used to generate C code from the specification. Copilot was demonstrated in numerous flight tests, including the first demonstration of Byzantine fault-tolerant RV [19]. The architecture of the older Copilot 2.0 framework is described in [18] along with a description of how lightweight formal methods were applied to the problem of monitor correctness. To address the challenge of ensuring that a formal specification is correct, research was conducted integrating model checking and SMT capabilities into the Copilot 2.0 framework [9,13].

Early incarnations of Copilot were very much research efforts and could be very clumsy to use in practice because, in embedded systems, many variables are stored as either C structs or C arrays, and Copilot could not handle these data structures. Given that we found no existing tool that supported generating code with arrays and structs, it was decided to build a new C code generator from scratch that could accommodate the needs of Copilot. This necessitated a considerable rewrite of the whole Copilot framework.

The decision to rewrite the framework corresponded with more stringent demands from users who wanted Copilot to generate monitors that were trustworthy enough to be certified by NASA as critical flight code. NASA began an initiative focusing on *high-assurance RV* [9]. The project adapted more structured software engineering processes following the NASA standard NPR7150.2C [16] that required extensive documentation such as documenting the architecture and design, software test plans, and coding style guides. In order to satisfy requisite assurance demands, unit tests were constructed for every module using Haskell's QuickCheck tool, and Galois Inc developed the

[1] NASA award NNL08AA19B, order NNL08AD13T.

Copilot Verifier that generates a mathematical proof that the generated monitor and specification are bisimilar [22]. In addition to the C code generator described in this tutorial, a prototype backend generating BlueSpec[2] has been developed enabling implementing monitors on FPGAs and exploratory work has begun on a Copilot backend for creating Rust monitors. Copilot was certified as a NASA Software Engineering tool (NPR7150.2, Class D) in June 2023 and Copilot developers continue to work with flight-safety groups at several NASA centers to improve the utility of Copilot for use in monitoring critical flight systems.

Today, Copilot continues to be developed as an open-source project,[3] with new releases being published every two months. Our Github repository contains detailed installation instructions for multiple operating systems. Users who wish to investigate how we meet some of the requirements of NPR7150.2 can visit our repository, specifically the issues, the pull requests, and the commit messages, where we leave evidence that can be used to audit our software development process. Users from the community are welcome to participate by providing contributions, asking questions, proposing new features, and by extending and using Copilot.

3 Background

Runtime verification is a dynamic software analysis technique that detects if a formally specified property is violated during an execution of a program or system. Copilot is not only a specification language, but a framework that transforms the specification into an executable monitor along with supporting code needed to observe the executing program. In the remainder of this section, we provide some background material that should aid understanding of RV in general and Copilot in particular.

3.1 Trace Theory

In order to check if an executing system satisfies a specification at runtime, the monitor must be able to observe a trace capturing the evolving state or events during execution. A *trace* [21] provides a view of an executing system that captures the evolving state or events that occur during a single run of a system. Suppose E is the set of states or events of an executing system and E^* is the set of all finite sequences of elements of E, then a trace $\tau \in E^*$ is a finite sequence of observed events or states. In RV, a trace is checked against a formal specification ϕ expressed in a formal logic. A specification denotes the set of traces that satisfy it. An RV monitor then must check that a trace τ is a member of the language of a specification ϕ, formally $\tau \in \mathcal{L}(\phi)$. This is sometimes phrased as "τ satisfies the specification ϕ" and expressed as $\tau \models \phi$. The SUO must be instrumented to extract the trace from the executing program and an RV framework should support the instrumentation.

[2] https://github.com/B-Lang-org.

[3] https://github.com/copilot-language/copilot.

3.2 Sampling

The idea of sampling representative data from a large set of data is well established in engineering. For instance, in digital signal processing, a signal such as music is sampled at a very high rate to obtain enough discrete points to represent the physical sound wave. The fidelity of the recording is dependent on the sampling rate.

Monitoring based on sampling state variables has historically been disregarded as a runtime monitoring approach, for good reason: without the assumption of synchrony between the monitor and observed system, monitoring via sampling may lead to false positives and false negatives [7]. For example, consider the property $(0, 1, 1)^*$, written as a regular expression, denoting the sequence of values a monitored variable may take (that is, in nominal conditions, we would expect the monitored variable to take the sequence of values of the shape $0, 1, 1, 0, 1, 1, 0, 1, 1, ...$). Depending on the specific times when the RV system samples the variable, both false negatives (the monitor erroneously rejects the sequence of values) and false positives (the monitor erroneously accepts the sequence) are possible. For example, if the actual sequence of values is $0, 1, 1, 0, 1, 1$, then an observation of $0, 1, 1, 1, 1$ will lead to a false negative, because a value has been skipped (Fig. 1). If the actual sequence is $0, 1, 0, 1$, then an observation of $0, 1, 1, 0, 1, 1$ will lead to a false positive, because a value is sampled twice (Fig. 2).

However, in a hard real-time context, sampling is a suitable strategy. Often, the purpose of real-time programs is to deliver output signals at a predictable rate. Under the assumption that the monitor and the observed program share a global clock and a static periodic schedule, false positives are possible, but false negatives are not. Moreover, in the context of cyber-physical systems, the data comes from sensors measuring physical attributes such as GPS coordinates, air speed, and actuation signals. Such continuous signals do not change abruptly and hence sampling suffices as long as it is performed at a suitable rate.

Many RV frameworks utilize inline monitors in the observed program to avoid the aforementioned problems with sampling. However, inlining monitors changes the real-time behavior of the observed program, perhaps in unpredictable ways. Introducing such unpredictability is not a viable solution for ultra-critical hard real-time systems. With Copilot's sampling-based approach, the monitor can be integrated as a separate scheduled process during available time slices (this is made possible by generating efficient constant-time monitors). Indeed, Copilot monitors may even be scheduled on a separate processor (albeit doing so requires additional synchronization mechanisms), ensuring time and space partitioning from the observed programs. Other RV frameworks targeting cyber-physical systems, such as R2U2 [12], have made the same design decision.

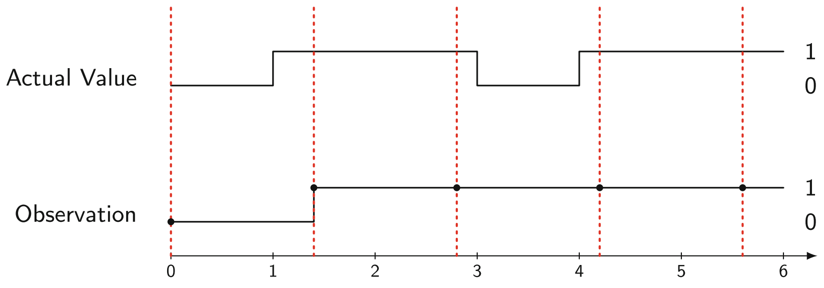


Fig. 1. Diagram representing values taken by a variable at regular points in time, and the observations taking place (at regular, but different, times), with a dotted line indicating the instants when the observations are taken. In this example, observations are made at a slower rate than values change, which may happen in a realistic scenario. In this case, the observation will lead to a false negative result compared to the actual value (the actual value is 0, 1, 1, 0, 1, 1, which conforms to the regular expression we are recognizing, whereas the observation is 0, 1, 1, 1, 1, which does not).

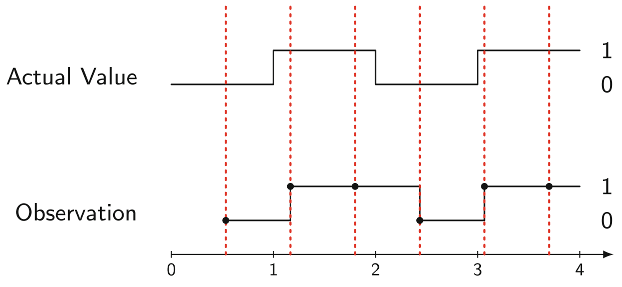


Fig. 2. Diagram representing values taken by a variable at regular points in time, and the observations taking place (at regular, but different, times), with a dotted line indicating the instants when the observations are taken. Notice that, in this example, the observation starts late compared to when the actual value is first set, which may happen in a realistic scenario. In this case, the observation renders false positive results compared to the actual value (the actual value is 0, 1, 0, 1, which does not conform to the regular expression we are recognizing, whereas the observation is 0, 1, 1, 0, 1, 1, which does conform to the regular expression).

4 Hello, Copilot!

The purpose of this section is to help the reader gain some intuition about the concepts that Copilot is built on, the different parts that make a monitor, and how the tool generates the code.

A Copilot monitor observes a system, analyzes the data being observed, and, if there are any property violations to report, executes functions that address those violations. Copilot does not determine how to fix the violations: its goal is to detect the problem and produce a notification.

Let us illustrate the main ideas in Copilot with a specific example of a collision avoidance system for a plane. When the plane is in cruise mode and governed by the autopilot, the altitude should not drop below a threshold, since that could indicate that there is a problem with the autopilot, the positioning system, or the sensors.

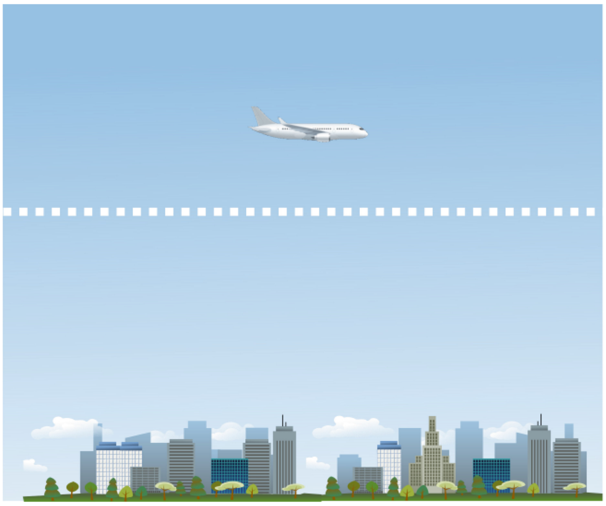


Fig. 3. Illustration of a plane flying over a city, with a dotted line indicating the minimum altitude at which the plane should be flying (not to scale).

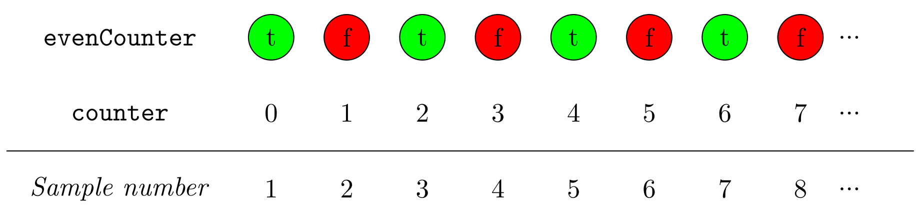
When writing a Copilot specification for a monitor, we must consider:

– What properties must be monitored.
– What data is needed i.e. the trace to be captured.
– What functions handle the monitoring violations, and what data is given to them.
– When the monitors are running.

In this particular example, the property that must be monitored is that the altitude of the aircraft is higher than a threshold when the autopilot mode is on. The data needed is 1) knowing whether the autopilot is on, 2) the altitude at any given time, and 3) the minimum altitude. In this example, we'll define input data streams for (1) and (2), and have a configuration parameter for (3) that remains constant throughout the flight. When the Copilot monitor detects that the system has entered an unsafe state, the monitor invokes an error handling function, which in this case will be called `recover`, providing as argument the current altitude and the desired threshold.

A Copilot monitor is an association between a boolean property and a handling function that is executed whenever the property becomes true. Monitors are to be checked at regular intervals or whenever new data is available.

To work with such changing values, Copilot sees data not as static, but as values that change over discrete time. Copilot properties can refer not just to the current values of the data, but also to past values. All data, from inputs to properties to outputs, are seen as streams, or infinite sequences of data samples. A property being monitored is represented by a time-varying boolean, also known as a boolean stream, but streams can carry other kinds of data (numeric, arrays, etc.). The following diagram depicts two streams: a numeric stream `counter`, which starts at zero and increases by one unit at every step, and a boolean stream `evenCounter`, which becomes true when the `counter` is even, and false otherwise.



At a glance, our monitor looks like this:

```
 1 import Language.Copilot
 2 import Copilot.Language
 3
 4 autopilot :: Stream Bool
 5 autopilot = extern "autopilot" Nothing
 6
 7 altitude :: Stream Word64
 8 altitude = extern "altitude" Nothing
 9
10 threshold :: Stream Word64
11 threshold = constant 10000 -- feet
12
13 property :: Stream Bool
14 property = autopilot ==> (altitude > threshold)
15
16 violation :: Stream Bool
17 violation = not property
18
19 monitor :: Spec
20 monitor = do
21   trigger "recover" violation [ arg altitude, arg threshold ]
```

There are several distinct parts to this specification. Lines 1–2 list libraries that must be imported. Lines 4–8 define two input streams: `autopilot` and `altitude`. The former is a stream carrying boolean values, and it is defined externally by an input by the same name. The latter is a stream carrying an unsigned 64-bit integer, and is also defined externally by an input by the same name. Lines 10–11 define an internal stream, also carrying unsigned 64-bit integers, which is constantly 10000.

Lines 13–17 define the desired system property: in this case, if the autopilot is on, the altitude should be greater than the threshold. The property is trivially true if the autopilot is off, as it is defined by an implication. The property that we wish to monitor is violated whenever it becomes false, as defined by `violation`.

Finally, in lines 19–21, the `monitor` is defined by a trigger that associates the handler `recover` to the stream `violation`, with two additional arguments. This means that, whenever violation becomes true, the external handler function `recover` will be called, passing the current values of `altitude` and `threshold` as arguments to the function.

Notice also the text `-- feet` in the definition of `threshold`, which represents a comment we include to help the reader understand the meaning of the constant.

5 The Copilot Specification Language

Copilot is a framework that comprises an RV specification language, and a tool that compiles specifications into C code. Copilot specifications are defined by a series of *triggers*, that is, properties that need to be monitored paired with functions that need to be called when those properties become true. Properties themselves are defined as Boolean-carrying *streams*, using a rich language of stream definitions that includes primitives and combinators, and gives access to external streams defined in C.

5.1 Streams

Streams are infinite successions of values, and constitute the central entity of Copilot specifications. Streams can be defined using primitives, or be built from other streams with a series of combinators. We provide a limited Application Programming Interface (API) to ensure that, by construction, streams are well-formed and can be compiled to efficient C code.

Constant Streams. The simplest `Stream` definition in Copilot is a constant stream of values, for which we provide the primitive `constant` that takes an element and returns a stream consisting of that element at every sample.

Example 1. The stream `true`, which Copilot defines in its Prelude for convenience, represents a constant stream carrying the Boolean value `True` and is defined as:

```
true = constant True
```

Because Copilot is a strongly typed domain-specific language, every expression and stream has a unique type. Following the notation of the host language Haskell, the type signature of every top-level definition is normally stated right before, prefaced by `::`. For example, `true`, as defined above, is a stream of Boolean values, which we denote with the type signature `true :: Stream Bool`.

The primitives and combinators that form the Copilot language are also functions that return streams. For example, the primitive `constant` itself is a Haskell function with type:

```
constant :: Typed a => a -> Stream a
```

The type signature of this primitive has two parts, separated by the symbol `=>`. On the right-hand side, the expression `a -> Stream a` indicates that `constant` is a function that takes an element of any type, which we call `a`, and returns a `Stream` carrying elements of the same type `a`. On the left-hand side, the expression `Typed a` is a *type constraint* and requires `a` to be an instance of the class `Typed`, denoting types that Copilot knows how to represent in C. We will not cover how to define custom types or instances in this document. Readers interested can consult standard Haskell textbooks to understand classes and instances, and the Copilot API to understand the type class `Typed`.

Including the type signatures explicitly is not always mandatory and it may be convenient to leave them out, especially when building very large and complex expressions. However, Copilot sometimes requires explicit type signatures for expressions that are ambiguous, in order to understand how to generate the corresponding C code. For example, the expression `constant 1` is a valid expression of type `Stream Int32`, but also one of type `Stream Int64` (and several other types). Without some type annotation, Copilot cannot know if it needs to use `uint32_t` or `uint64_t` in C to store the data, and so it requires that we spell out the type of the expression. To minimize the need for type annotations, Copilot provides a family of constant stream-building functions for each primitive type. For example, we can define the constant stream of 1's using 64-bit integer numbers as:

```
ones :: Stream Int64
ones = constI64 1
```

In this case, the type signature is redundant: because we have used the primitive `constI64`, the compiler knows we mean to build a stream of 64-bit integers. Conversely, we could have instead defined this stream using the 'constant' primitive; since we explicitly state the type of the stream, the compiler knows which representation to use. For streams carrying numbers, we can also use the literal number without 'constant'or 'constI64'. The compiler knows that literal numbers just mean constant streams whenever a stream is expected.

Lifting and Point-Wise Function Application. We provide definitions that extend the standard API of each type supported by Copilot to act pointwise on streams. Copilot supports Boolean values (i.e., `Bool`), signed and unsigned fixed-length integers (i.e., `Word8`, `Word16`, `Word32`, `Word64`, `Int8`, `Int16`, `Int32`, `Int64`), floating point numbers (i.e., `Float`, `Double`), limited structs, and limited arrays. The operators and combinators provided by Copilot are limited to a subset that we can compile to C efficiently. Details on structs and arrays are given in Sects. 5.2 and 5.3 respectively.

Example 2. The standard negation function `not`, operating on Booleans, would normally take one Boolean value and return another Boolean. Copilot defines `not` to operate on *streams of Booleans.* For example, the stream `false`, which holds the constant value `False`, could be defined by applying `not` to negate every value in the `true` stream defined earlier:

```
false :: Stream Bool
false = not true
```

Streams can contain other values representable in C, like integers and doubles. We overload literal numbers and mathematical operators to work on streams: literals denote constant streams, and operators are applied pointwise. For example, we can define the constant stream carrying the number 4 based on the definition of `ones` from before, as:

```
fours :: Stream Int64
fours = ones + 3
```

In this definition, the symbol `3` denotes the constant stream of `3`'s, and the symbol `+` denotes addition of streams carrying numbers, defined pointwise (e.g., the first element of `ones` plus the first element of `3`, the second element of `ones` plus the second element of `3`, and so on).

Temporal Translations. Because streams represent values that change over time, it seems natural to think of how to refer to a past or future value of a stream. Copilot provides two mechanisms to translate a stream in time: delays, which allow us to refer in the present to values in the past, and drops, which allow us to refer in the present to values in the future. Both impose additional constraints to ensure that the resulting specification is well-defined, and that it can be executed in real time.

Delaying a stream requires that we hold the stream's actual value in memory for future use. Unbounded delays (i.e., those in which the amount of elements to hold is potentially unbounded) are known to lead to memory leaks [5]. To make the generated C code efficient and memory usage predictable, we provide very limited ways of delaying streams: streams can be delayed by pre-pending *a fixed number of samples*, with the operator `(++)`, with type:

```
(++) :: Typed a => [a] -> Stream a -> Stream a
```

Example 3. We can use the append operator `(++)` to create a stream that is initially `False` and later becomes `True` indefinitely:

```
falseThenTrue :: Stream Bool
falseThenTrue = [False] ++ true
```

Note that streams can be defined recursively. For example, we can define the stream that alternates between the values `True` and `False` as:

```
alternatingStream :: Stream Bool
alternatingStream = [True] ++ not alternatingStream
```

Using recursion, like before, we can define a step counter (e.g., $[1, 2, 3, \ldots]$) as follows:

```
counter :: Stream Int32
counter = [1] ++ (counter + 1)
```

Copilot also provides the opposite temporal transformation, dropping elements from a stream, with the function:

```
drop :: Typed a => Int -> Stream a -> Stream a
```

Example 4. In the following example, we use drop to eliminate the first 2 elements from a stream:

```
numbers :: Stream Int64
numbers = [1,34,2,9,8,15] ++ numbers

numbers' :: Stream Int64
numbers' = drop 2 numbers   -- 2, 9, 8, 15, 1, 34, 2, 9,...
```

Dropping elements introduces a potential issue if the elements are not available, which may happen if they come from an external source (e.g., a sensor). This is discussed in the following.

External Streams. To connect Copilot specifications to existing C applications, we provide the primitive `extern` to define a stream based on the value of a global C variable, by indicating its name and its type. Within Copilot, we have no way of guaranteeing that a given variable exists, or that it has the expected type. However, from a specification containing an external stream, Copilot generates a C header file that declares the existence of an extern global variable with a specific type. The use of a variable name that does not exist, or that has the wrong type, would give rise to either an error or a warning when trying to compile and link the generated C code as part of a larger application.

Example 5. Commonly in Copilot specifications, there is a need to access data provided by an external sensor. For example, given a global variable `altitude`, of a type `Word64`, holding the current altitude of the plane, we can define a Copilot specifications as follows:

```
altitude :: Stream Word64
altitude = extern "altitude" Nothing
```

The additional argument `Nothing` contains an optional list of `Word64`s that can be used during simulation, when actual data from the sensor is not available.

External streams are one example of a stream from which we cannot drop samples, since that would require being able to provide data that has not been produced by the system yet. If we try to drop samples from a stream, for example, by using the expression `drop 1 altitude` anywhere in our specification, Copilot reports a *compile-time* error:

```
Copilot error: Drop applied to non-append operation!
```

The error is not reported if we first append samples to the stream, for example, with `drop 1 ([a1, a2] ++ altitude)` (where `a1` and `a2` are two valid and known altitudes). If we drop more samples than we prepend, however, Copilot still reports an error:

```
Copilot error: Drop index overflow!
```

Definitions that drop "too much" are disallowed because they present potential violations of causality: the present of a stream may depend on a future that has not happened yet. Problems with non-causal definitions are common in temporal frameworks [1,8]. By introducing the afforementioned checks, Copilot is able to prevent potential causality errors at compile time.

5.2 Structs

Copilot structs are compiled into C structs and made available to the monitor. To be able to generate a correct `struct` definition in C, Copilot structs need to be defined using specific Copilot types. We normally implement structs in Copilot as records made of *fields*, each having a *name* and a *type*. We use the type `Field s t` to represent each field, with `s` being the field name (a type-level literal string), and `t` being the type of the field.

Example 6. To demonstrate how to work with structs, let us use a different example inspired by one of the properties we monitor in our systems: the temperature of a battery in one of the aircraft's components. The following defines a new Copilot type `Battery` with a field `temperature` of type `Int16`,[4] which corresponds to a struct in C with a field `temp` of type `int16_t`:

```
data Battery = Battery
  { temperature :: Field "temp" Int16 }
```

[4] For the sake of simplicity, we omit other fields in the definition of `Battery`.

We provide a limited API to operate on streams of structs or records. Currently, Copilot supports projections, that is, accessing a field of a struct, with the function:[5][6]

```
(#) :: (Typed a, Typed t)
    => Stream a -> (a -> Field s t) -> Stream t
```

The first argument denotes the stream carrying a struct of type `a`, and the second denotes a field of the struct with name `s` and type `t`. Because structs are first class, they are valid types to be used in streams, and so are their fields.

Example 7. If we have a global C variable `battery` of the struct type generated from the definition of `Battery` above, holding the state of the battery at each step, we can access it from Copilot with the following definition:

```
batt :: Stream Battery
batt = extern "battery" Nothing
```

We can extract a field of a stream of structs, producing another stream in a type-safe way:

```
tempPlus1 :: Stream Int16
tempPlus1 = batt # temperature + 1
```

5.3 Arrays

Copilot also includes support for arrays, with an advanced type that includes the length as part of the type of the array. For example, a stream in which each element is an array with 16 elements of type `Int64` would have type `Array 16 Int64`. The presence of the array's length as a type-level natural number serves two purposes: first, it allows the compiler to detect, at compile time, some incorrect accesses (i.e., access out of bounds), and, second, it allows us to generate C without dynamic memory allocation, as all arrays have known, fixed lengths.

Similarly to structs, Copilot provides a limited API to work with `Stream`s of `Array`s. To access specific elements in the array, we provide the operator[7]:

```
(.!!) :: (KnownNat n, Typed t)
      => Stream (Array n t)
      -> Stream Word32
      -> Stream t
```

[5] The signature of `(#)` includes additional constraints. We omit details out of brevity, but this does not change the way that it is used.

[6] The parentheses around some operator names are a particularity of the host language, but they are not needed when the operators are used in infix position in expressions, as illustrated later in this section.

[7] The signature of `(.!!)` imposes additional constraints on the array, which we omit, but the arguments and the way they are used are as described.

This operation allows us to access an element of an array, where the position of that element is determined by a number in a stream. The signature of this operator includes a requirement that `n` be a `KnownNat`. That is just a fancy way of saying that it must be a concrete number known at compile time, that is, a specific natural number (as opposed to a variable holding a natural number).

Example 8. We can augment `Battery` with the measurements of the voltages of individual cells by including a new field:

```
data Battery = Battery
  { temperature :: Field "temp"  Int16
  , voltages    :: Field "volts" (Array 10 Word16)
  }
```

The field `voltages` is an array of length 10, whose elements have type `Word16`.

Copilot allows us to define streams that access specific elements of these arrays. For example, we can take the `voltages` field from `batt`, extract its first element, and add one to the result.

```
volt0Plus1 :: Stream Word16
volt0Plus1 = (batt # voltages .!! 0) + 1
```

Copilot is able to detect some incorrect accesses at compile time, if the index within the stream is out of range and the Copilot expression denoting that index is a constant stream. For example, if we had passed 11 as second argument of `(.!!)`, we would have seen a warning during compilation. Nevertheless, it is not generally possible for all streams to detect incorrect accesses in compile time, since the value of the stream containing the index may only be determined at runtime if they depend on some external variable.

5.4 Monitors

The purpose of Copilot is to monitor properties and to raise an alert when an assertion is violated. The Copilot language defines monitors as sequences of *triggers*. A trigger is defined as a stream of Booleans, a C function to be called when the current sample is true, and the arguments to pass to that function:

```
trigger :: String -> Stream Bool -> [Arg] -> Spec
```

`Spec` is a type internal to Copilot and represents a specification.

Properties in triggers denote violations, not assertions. Therefore, triggers denote functions to call when samples are `True`, not `False`.

The function to call is given by the first argument as a `String`, and needs to be implemented by the user. Referring to a function that does not exist would lead to a link-time error. If the header files generated by Copilot are included in other C files of the application that uses Copilot, referring to a function with the wrong arguments in a trigger would also lead to a compilation error.

Example 9. The following specification declares a monitor that executes the C function `large`, passing as argument the current value of the stream `counter`, when the voltage of the first cell of the battery, plus one unit, is greater than 8. Arguments are passed as a comma-separated list, with the keyword `arg` preceding each argument stream:

```
monitor :: Spec
monitor =
  trigger "large" (volt0Plus1 > 8) [arg counter]
```

`Spec`s represent computations, so we can declare multiple triggers by listing them in sequence, preceded by the language keyword `do` (what is known in the host language Haskell as `do` notation).

We can expand the prior definition by adding a second trigger that calls the function `too_large` with no arguments when the same voltage is greater than 10 (see Sect. 5.3):

```
monitor :: Spec
monitor = do
  trigger "large" (volt0Plus1 > 8) [arg counter]
  trigger "too_large" (volt0Plus1 > 10) []
```

Copilot specifications can be simulated on a computer, or compiled into C code to be used in the same or a different device. In Sect. 7 we demonstrate how the specification can be compiled into C and integrated in a larger system.

6 Logics and Languages

Monitors and specifications can become overly complex as systems grow. To aid understanding, Copilot supports extending the language with new operators without having to modify its internals. Users can leverage such facilities to simplify their specifications and reuse constructs across projects.

This section introduces the temporal and propositional logic libraries of Copilot, which are defined using the aforementioned extension mechanisms. In this tutorial we concentrate on the past-time temporal logics. However, Copilot also includes libraries for future-time linear temporal logics, including bounded LTL and metric temporal logic.

6.1 Logical Operators

As mentioned in Sect. 5, Copilot extends the standard APIs of the supported types to apply pointwise on streams. In the case of Booleans, Copilot provides a number of logical operators based on propositional logic. Apart from the constants `true` and `false`, the following operators on Boolean streams are provided:

```
not   :: Stream Bool -> Stream Bool
(&&)  :: Stream Bool -> Stream Bool -> Stream Bool
(||)  :: Stream Bool -> Stream Bool -> Stream Bool
xor   :: Stream Bool -> Stream Bool -> Stream Bool
(==>) :: Stream Bool -> Stream Bool -> Stream Bool
```

In all cases, these operators apply the associated Boolean operation to the values at each sample. For example, given two Boolean streams `s1` and `s2`, the stream `s1 ==> s2` is true at a time (i.e., sample) if `s2` is true at that time, or if `s1` is false.

While these logical operators can help simplify basic expressions, the complexity of real-world applications demands higher-level languages. In the following we explore temporal logics supported by Copilot, and introduce operators to refer to past or future values of a stream.

6.2 Temporal Logics

Generally speaking, temporal logics extend other logics with a temporal dimension. To describe relations between formulas at different times, temporal logic languages introduce modal operators that abstract over time. For example, some languages provide an operator $\Box$ (called *always*, also written G, after the word *globally*), and a formula $\Box\ \phi$ is true if and only if ϕ is true at all times, where the specific meaning of the expression "at all times" depends on the logic.

Temporal logic languages generally vary in the logic they are based on, the temporal operators they support and in their model of time (e.g., continuous vs discrete, linear vs branching, future and/or past). These aspects impact what formulas can be expressed, which ones are true or false, what information is needed to evaluate them, and how efficiently we can do so.

Because time is an essential component of stream languages, like Copilot, temporal logics constitute a suitable mechanism to express many of the re-occurring patterns in monitor specifications. In the following we discuss some of the languages supported by Copilot, and demonstrate their use with examples.

Past-Time Linear Temporal Logic. Past-time Linear Temporal Logic (ptLTL) is an extension of propositional logic in which time is seen as linear, discrete, and bounded. While, in propositional logic, every variable may take the value true or false, in ptLTL, every variable may take the value true or false, *at each point in the present or in the past.*

Past-time linear temporal logic introduces temporal operators, letting us express logical formulas based not only on certain propositions being true at the present, but also on their validity in the past.

Copilot supports the temporal operators `alwaysBeen`, `eventuallyPrev`, `previous`, and `since`, all operating on and returning Boolean streams:

```
alwaysBeen     :: Stream Bool -> Stream Bool
eventuallyPrev :: Stream Bool -> Stream Bool
```

```
previous        :: Stream Bool -> Stream Bool
since           :: Stream Bool -> Stream Bool -> Stream Bool
```

A stream `alwaysBeen x` is true at a time if `x` has been true at all times, present and past (Fig. 4). The operator `eventuallyPrev` works the opposite way, and `eventuallyPrev x` is true if `x` has ever been true. The temporal operator `previous` refers to the immediately previous sample, with `previous x` being true if `x` was true in the last sample. Finally, `since x y` is true at a time if the stream `x` has been continuously true since the first sample after `y` became true.

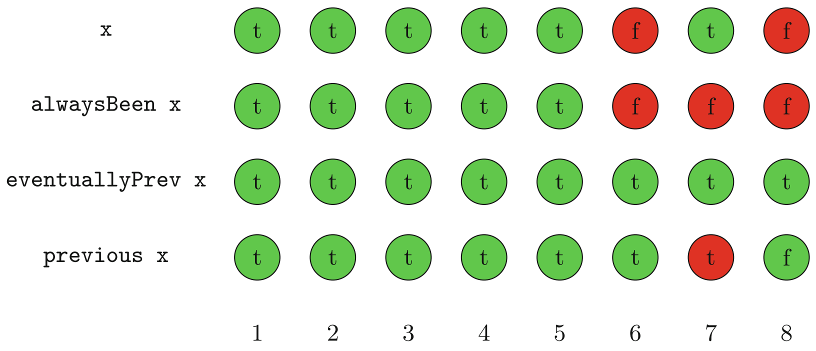


Fig. 4. Example of values of the formulas `alwaysBeen x`, `eventuallyPrev x`, and `previous x`, for different values of `x` at different times.

Example 10. Borrowing the example in the prior section, imagine that we want to detect if the voltage of the first battery cell was ever too high. We can express that in Copilot with the following specification:

```
voltageWasTooHigh :: Stream Bool
voltageWasTooHigh = eventuallyPrev ((batt # voltages .!! 0) >= 10)
```

We can combine these temporal operators with the pointwise operators presented earlier in this section, to capture, for example, that a safety system can only be activated if a condition was violated before:

```
safetyResponseOK :: Stream Bool
safetyResponseOK = safetyEngaged ==> voltageWasTooHigh
  where
    safetyEngaged :: Stream Bool
    safetyEngaged = extern "safety_system" Nothing
```

7 Integration Into Larger Systems

The facilities described until now allow users to specify properties to be monitored. To use such monitors in practice, users need to provide the inputs to the monitors and define the functions to handle property violations.

This section describes how to generate the C code for the monitors, and how to connect the monitors to the rest of the system under observation. The purpose is to give the reader a full understanding of the interfaces used by Copilot. However, for larger projects and existing software frameworks, we have developed a separate tool that simplifies the process. Currently, the tool supports NASA's Core Flight System (cFS) [23], FPrime [3], the Robot Operating System 2 (ROS 2), and we welcome contributions to support other platforms.

7.1 Monitor Generation in C

To generate the C code for the monitors, we need to modify the specifications to indicate that we wish to compile the specification to C. Taking the example from Sect. 4 about the altitude / autopilot as a starting point, we first need to add one more import, namely:

```
import Copilot.Language.C99
```

We can now add a main function that compiles the specification. We first need to reify it, which in Copilot's compiler is a process that also checks the specification's correctness. The result of the reification process is a post-processes specification that we can compile to C:

```
main = do
  reifiedSpec <- reify monitor
  compile "rv" reifiedSpec
```

To actually compile the spec and generate the C code, all we need to do is run the specification. Because Copilot is a DSL embedded in Haskell, we use the standard tool `runhaskell` (assuming that the specification is saved in a file called `Spec.hs`):

```
$ runhaskell Spec.hs
```

The result of this execution are three files, namely `rv.c`, `rv.h`, and `rv_types.h`. The first file contains the actual implementation of the monitors. We do not need to concern ourselves with how the file is implemented; everything we need to interact with the monitors is declared in `rv.h`. The latter file `rv_types.h` contains local definitions of any auxiliary types used in the specifications (e.g., structs). Since in this case there are none, that file is empty.

Inspecting `rv.h` more closely shows a number of definitions that we need to provide, or that we can use:

```
extern bool autopilot;
extern uint64_t altitude;
void step();
extern void recover(bool arg1, uint64_t arg2);
```

The first two declarations are the inputs to the system. It is the Copilot user's obligation to define those somewhere and give them appropriate values at all times and before the monitoring system checks the current state of the monitors. The next line, `void step();`, declares the main entry point of the monitoring system. The user of this code must call the `step` function when they want to check the status of the monitors. Note that calling `step` also advances the implicit clock for the monitoring system. Finally, the function `recover` must be defined by the user of this code, and must implement a mechanism to recover from the property violation.

7.2 Integration

The following is a sample C code that puts data in both inputs, calls `step`, and prints messages whenever there are violations:

```c
bool autopilot = false;
uint64_t altitude = 0;
void recover(bool arg1, uint64_t arg2) {
  printf("Violation: %d\n", arg2);
}
int main () {
  int i = 0;
  for (i = 0; i < 10000; i++) {
     autopilot = (i % 1000) < 500;
     altitude  = abs (i - 500);
     step();
  }
}
```

In this code, the `recover` function simply reports the violation, but does not actually recover from the off-nominal situation. However, in a real system, the `recover` function could activate a recovery routine that actually brings the plane to a higher altitude. In general, the details of the recovery routine are mission specific and quite complex. If, for example, we were working with a quadcopter instead of a plane, the recovery routine for a loss of altitude might increase the power on the propellers, provided that the quadcopter is level, rightside up, and there are no objects in its path moving upwards, and other necessary safety conditions hold. Other recovery methods for the same situation would be possible, depending on the vehicle and the situation.

8 Conclusion

In this paper, we described Copilot, a runtime verification framework for safety-critical, real-time embedded systems. We discussed the evolution of the project, how it was originally built, and how the components that make the framework have changed over time based on both project need and resources available. The current version of the language was introduced, and we saw how the new constructs of structs and arrays help to deal with more complex data structures without sacrificing safety. Copilot is a project rich in libraries, and we discussed different temporal logics supported by the language. We also illustrated how to create monitors and integrate them into an existing system with minimal interference with the SUO.

There are a number of open problems in this domain, and in Copilot as a whole, that remain to be addressed. We expect future versions of the language to be simpler and require less boilerplate code, to support generating monitors in languages other than C, and to expose less of the underlying Haskell ecosystem.

Acknowledgements. Figure 3 is based on image "City skyline" by Rg1024 (public domain), and image "Large modern passenger airliner jet", Designed by macrovector/Freepik.

References

1. Bahr, P., Berthold, J., Elsman, M.: Certified symbolic management of financial multi-party contracts. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, pp. 315–327. ICFP 2015, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2784731.2784747
2. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification, pp. 1–33. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
3. Bocchino, R., Canham, T., Watney, G., Reder, L., Levison, J.: F Prime: an open-source framework for small-scale flight software systems (2018)
4. Caspi, P., Pialiud, D., Halbwachs, N., Plaice, J.: LUSTRE: a declarative language for programming synchronous systems. In: 14th Symposium on Principles of Programming Languages, pp. 178–188 (1987)
5. Courtney, A., Elliott, C.: Genuinely functional user interfaces. In: Haskell Workshop, pp. 41–69 (2001)
6. D'Angelo, B., et al.: LOLA: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning, pp. 166–174. IEEE (2005)
7. Dwyer, M., Diep, M., Elbaum, S.: Reducing the cost of path property monitoring through sampling. In: Proceedings of the 23rd International Conference on Automated Software Engineering, pp. 228–237 (2008)
8. Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, pp. 263–273. ICFP '97, ACM (1997)
9. Goodloe, A.: Challenges in high-assurance runtime verification. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10–14, 2016, Proceedings, Part I, pp. 446–460 (2016)
10. Goodloe, A., Pike, L.: Monitoring distributed real-time systems: a survey and future directions. Tech. Rep. NASA/CR-2010-216724, NASA Langley Research Center (2010)
11. Havelund, K., Goldberg, A.: Verify your runs. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments, pp. 374–383. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69149-5_40

12. Johannsen, C., Jones, P., Kempa, B., Rozier, K.Y., Zhang, P.: R2U2 Version 3.0: re-imagining a toolchain for specification, resource estimation, and optimized observer generation for runtime verification in hardware and software. In: Enea, C., Lal, A. (eds.) Computer Aided Verification, pp. 483–497. Springer Nature Switzerland (2023)
13. Laurent, J., Goodloe, A., Pike, L.: Assuring the guardians. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification, pp. 87–101. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_6
14. Knight, J.C.: Safety critical systems: challenges and directions. In: Proceedings of the 24th International Conference on Software Engineering, pp. 547–550. ICSE '02, ACM (2002)
15. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, New York, NY (1992). https://doi.org/10.1007/978-1-4612-0931-7
16. NASA: NASA Software Engineering Requirements NPR7150.2C. https://nodis3.gsfc.nasa.gov/displayAll.cfm?Internal_ID=N_PR_7150_002C_ (2019)
17. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: a hard real-time runtime monitor. In: Barringer, H., et al. (eds.) Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings, pp. 345–359. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_26
18. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Experience report: a do-it-yourself high-assurance compiler. In: Proceedings of the International Conference on Functional Programming (ICFP). ACM (2012)
19. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Copilot: monitoring embedded systems. Innov. Syst. Software Eng. **9**(4) (2013)
20. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. SFCS '77, IEEE Computer Society, Washington, DC, USA (1977). https://doi.org/10.1109/SFCS.1977.32
21. Reger, G., Havelund, K.: What is a trace? A runtime verification perspective. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 339–355. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_25 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016); Conference date: 05-10-2016 Through 14-10-2016
22. Scott, R.G., Dodds, M., Perez, I., Goodloe, A.E., Dockins, R.: Trustworthy runtime verification via bisimulation (experience report). Proc. ACM Program. Lang. **7**(ICFP) (2023)
23. Wilmot, J.: A core flight software system. In: Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 13–14. CODES+ISSS '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1084834.1084842

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

ASMETA Tool Set for Rigorous System Design

Andrea Bombarda[1](✉), Silvia Bonfanti[1], Angelo Gargantini[1], Elvinia Riccobene[2], and Patrizia Scandurra[1]

[1] University of Bergamo, Bergamo, Italy
{andrea.bombarda,silvia.bonfanti, angelo.gargantini,patrizia.scandurra}@unibg.it
[2] Università degli Studi di Milano, Milan, Italy
elvinia.riccobene@unimi.it

Abstract. This tutorial paper introduces ASMETA, a comprehensive suite of integrated tools around the formal method Abstract State Machines to specify and analyze the executable behavior of discrete event systems. ASMETA supports the entire system development life-cycle, from the specification of the functional requirements to the implementation of the code, in a systematic and incremental way. This tutorial provides an overview of ASMETA through an illustrative case study, the Pill-Box, related to the design of a smart pillbox device. It illustrates the practical use of the range of modeling and V&V techniques available in ASMETA and C++ code generation from models, to increase the quality and reliability of behavioral system models and source code.

1 Introduction

It is widely recognized that formal methods need to be supported by automated tools to be of practical use and promote their adoption, especially when they are required by critical application areas (such as security and safety) and standards for software certification and accreditation [6,7,22,23]. This tutorial presents ASMETA[1], an open-source framework defining modeling notations and tools inspired by the well-known formal method of the Abstract State Machines (ASMs) [14,15]. ASMETA supports model editing, visualization, simulation, animation, validation, verification, as well as code generation from formal models.

In the wide range of existing formal methods [17,21], and more specifically of state-based formal methods[2], the ASM-based formal method supported by

[1] https://asmeta.github.io/.
[2] https://abz-conf.org/methods/.

The work of Andrea Bombarda is supported by PNRR - ANTHEM (AdvaNced Technologies for Human-centrEd Medicine) - Grant PNC0000003 - CUP: B53C22006700001 - Spoke 1 - Pilot 1.4. The work of Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

© The Author(s) 2025

ASMETA offers several advantages: (1) models have a *pseudo-code format*, so practitioners can easily understand them as *high-level programs*; (2) systems can be specified at any desired *level of abstraction*, depending on the level of details one wants to achieve; (3) models are *executable*, so they are suitable also for lighter forms of model analysis such as simple simulation to check model consistency w.r.t. system requirements; (4) techniques for mapping models to code (e.g., to C++ or Java) are supported, so *correct-by-construction* development is possible; (5) *multi-agents modeling* is supported, making possible the specification of *distributed systems*. Moreover, the ASMETA framework allows for the integrated use of tools for different forms of model analysis, it is maintained and under continuous features improvement.

Through an illustrative case study from the healthcare domain, the Pill-Box system, this tutorial shows how to model in ASMETA the executable behavior of a system with a discrete state space. Then, the tutorial guides the readers through the use of the ASMETA tools to apply several model validation and verification (V&V) techniques, such as simulation, scenario-based validation, and formal verification of user-defined properties and meta-properties. A model refinement process supported by ASMETA is also presented by means of the running case study and by explaining how in formalizing the system behavior it is possible to evolve a partial specification (ground model) into a more complete model. Finally, the tutorial showcases the automatic generation of executable C++ code from the Pill-Box model, developed and verified in ASMETA.

This tutorial is intended to be a resource for software engineers and researchers that want to leverage lightweight formal methods in their projects. The hands-on approach, adopting the Pill-Box system as running example, endows the readers with the necessary skills to start adopting ASMETA for a more rigorous system design that increases the quality and reliability of behavioral system models and source code. ASMETA is distributed as an open-source solution so that other researchers can contribute to its extension.

The remainder of this tutorial is organized as follows. Section 2 introduces the ASMs, while Sect. 3 introduces the ASMETA framework together with a modeling process, and provides all useful references. Section 4 presents the running case study. Section 5 describes the user-facing modeling language `AsmetaL` to define ASM models. Sections from 6 to 9 explain the ASMETA tooling supporting all model analysis techniques for a rigorous system design. Section 10 presents model refinement applied to the running case study. Section 11 explains how to generate C++ code from the verified Pill-Box model. Finally, Sect. 12 concludes.

2 Abstract State Machines

Before introducing ASMETA, here we provide a basic introduction of the state-based formal method of ASMs [14,15]. *States* are mathematical *algebras* specifying a system configuration by means of arbitrarily complex data, i.e., domains of elements with functions defined on them. State *transitions* are expressed by named and parameterized transition rules describing how the data (function values saved into *locations*) change from one state to the next one.

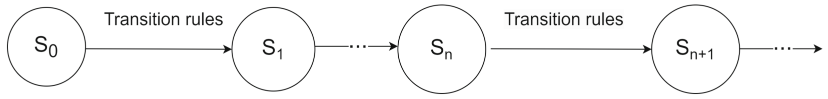


Fig. 1. An ASM run with a sequence of states and state-transitions (steps)

The functions of the algebra are classified into *dynamic* and *static* depending on whether they are updated or not by transition rules. The dynamic functions are further distinguished in *monitored* (read by the machine and modified by the environment), *controlled* (read and written by the machine), *out* (only written by the machine and read by the environment), and *shared* (read and written by the machine and its environment). In addition, functions that are defined in terms of other (dynamic) functions are called *derived*.

Dynamic functions are updated by firing transition rules. The basic transition rule is the *update* rule for function update; it has form $f(t_1,\ldots,t_n) := v$, where f is an n-ary function, t_i with $i = 1..n$ are terms, and v the new value to be associated with the location $f(t_1,\ldots,t_n)$ in the next state. As in structured programming, constructs for structured control flow can be used to form transition rules depending on the type of update structure they express. The main rule constructors include: guarded updates (*if-then*, *switch-case*), simultaneous parallel updates (*par*), non-determinism (*choose*), unrestricted synchronous parallelism (*for-all*), abbreviation on terms of rules (*let*), etc.

ASMs can be read as pseudocode over abstract data with a well-defined execution semantics. An ASM *run* (see Fig. 1) is a (finite or infinite) sequence $S_0, S_1, \ldots, S_n, \ldots$ of states. Starting from the initial state S_0, in a computation step (*run step*) from S_n to S_{n+1}, all enabled transition rules are executed in parallel, leading to simultaneous updates of a number of locations. In case of an inconsistent update (i.e., the same location is updated to two different values) or invariant violations (i.e., some property that must be true in every state is violated), the model execution stops with error.

3 Overview of the ASMETA Toolset

The ASMETA project started in 2004 with the aim of addressing the deficiency in tools that support ASMs. Although the formal approach had demonstrated widespread application in specifying and verifying various software systems across diverse domains (as evidenced by the ASM research summary in [15]), the absence of supportive tools for the ASM method was deemed a limitation, leading to skepticism regarding its practical utility.

To address this issue, ASMETA has been developed by exploiting the Model-Driven Engineering (MDE) approach [5] for software development starting from the definition of a meta-model for an abstract notation able to capture the *working definition* (see [15, pag. 32]) of an ASM. From the metamodel, a textual notation for encoding ASM models has been derived, and has been enriched, during the years, to support many V&V activities in the rigorous design of software systems. These analysis techniques have been proven to be beneficial for the safety assurance of safety-critical systems with event-based behavior and discrete state spaces. See [1] for further details on the case studies and application domains (including medical software, software control systems, and service-based systems, to name a few) to which ASMETA has been applied.

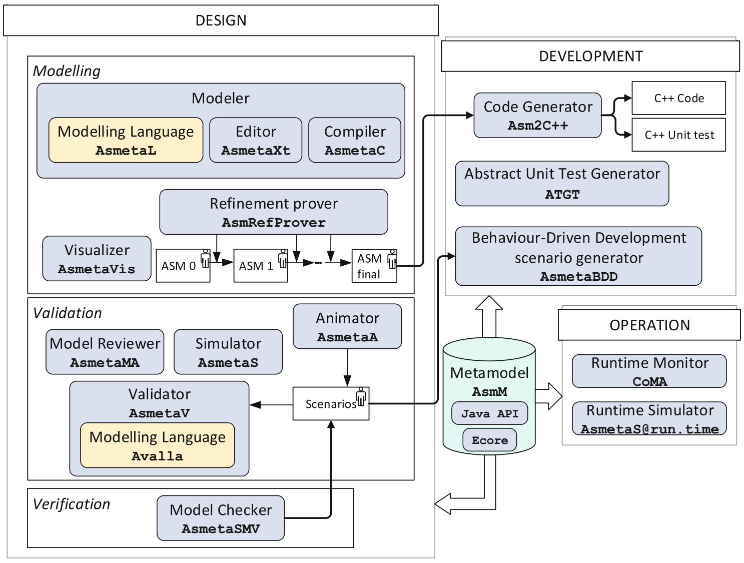


Fig. 2. ASMETA-based development process

3.1 Getting and Using ASMETA

Most ASMETA tools are integrated with the Eclipse IDE[3]. An Eclipse package containing the ASMETA toolset is available and released periodically at https://github.com/asmeta/asmeta. In the same location, the source code of all the ASMETA tools, together with examples of ASMETA specifications, is available.

Tooling. ASMETA tools support the main activities of the software development process from formal requirement specification to code generation. Figure 2 shows the tools usage in the various stages [1]. At *design* time, ASMETA provides tools for model editing and visualization (the modeling language `AsmetaL`[4] and its editor and compiler, plus the model visualizer `AsmetaVis` for graphical visualization of ASMETA models), model validation (e.g., interactive or random simulation by the simulator `AsmetaS`[5], animation by the animator `AsmetaA`, scenario construction and validation by the validator `AsmetaV`, and static analysis

[3] An Eclipse package including all tools and models useful for this tutorial is available at https://doi.org/10.5281/zenodo.12770854.

[4] It is a concrete notation for the abstract one defined by the metamodel reflecting the working definition of an ASM.

[5] The ASMETA model simulator implements the computational paradigm (concepts and semantics) of an ASM run as defined in the previous section.

by the model reviewer `AsmetaMA`), and verification (proof of temporal properties by the model checker `AsmetaSMV`, and proof of correct model refinement by `AsmRefProver`). During software *development*, ASMETA supports automatic code and test case generation from models (the code generator `Asmeta2C++`, the unit test generator `ATGT`, and the acceptance test generator `AsmetaBDD` for complex system scenarios). If the system is available, during its *operation*, ASMETA can be used for runtime monitoring (by the tool `CoMA`) and runtime simulation (by `AsmetaS@run.time`).

Remark. Due to lack of space and to keep this tutorial simple and understandable to new and unfamiliar users, in the following sections we explain and show the application of a selected number of tools, those supporting the initial and fundamental steps of system modeling, analysis (V&V) and encoding. Focusing more on pedagogical rather than technical aspects of our modeling approach, we also skip advanced modeling features (e.g., the concepts of multi-agent ASMs or I/O ASMs, suitable to model distributed and composable systems) which require understanding the basic and preliminary concepts around ASMETA; this is what this tutorial intends to cover.

Modeling Process. ASMETA derives its foundation from the ASM theory, thus, akin to ASMs, its modeling methodology follows an iterative approach with a focus on model refinement. Concretely, ASMETA employs *stuttering refinement* [4], a specialized variant of the broader ASM refinement [13]. This refinement-based process allows users to tackle the complexity of the requirements and to bridge, in a seamless manner, specification to code. Requirements modeling begins with the creation of a high-level ASMETA model, akin to the ASM ground model [15]. This model is delineated through the analysis of informal requirements typically presented in natural language. Model signature and rule naming are set by using terms of the application domain and derived from textual requirements, thereby simplifying the process of connecting requirements to the model. This high-level model (see model ASM_0 in Fig. 2) should be *correct* and *consistent*, i.e., it should represent the intended requirements (at the desired abstraction level) and no ambiguities of initial requirements should be left. It is not necessary for ASM_0 to be *complete*, i.e., it may not specify some given requirements that are later captured during the refinement process. Indeed, the modeling process supported by ASMETA is a refinement-based one: starting from the model ASM_0, through a sequence of *refined* models ASM_1, ASM_2,..., other functional requirements are specified and modeled, till the desired level of completeness is reached. At the end of this process, $\mathsf{ASM}_{\text{final}}$ captures all intended requirements at the desired level of abstraction. When performing refinements, it is important to prove that each refined model is a correct (stuttering) refinement of the previous one. The ASMETA framework includes the `AsmRefProver` tool [4] which supports the user in this activity and automatically performs the correctness check of refinement steps.

Starting from the very first model, the ASM_0, during the modeling process the user should perform validation and verification (V&V) activities to assure requirements satisfaction and property validity.

4 The Pill-Box Case Study

In this section, we introduce the Pill-Box case study [8] with its informal requirements, which will be used throughout the paper as a running example to describe the modeling and analysis activities supported by the ASMETA tools.

The Pill-Box device is a medicine/pill dispenser that has a certain number of drawers (e.g., three drawers). Each drawer contains multiple slots (one for each pill) that are emptied in sequence. In each drawer, only one specific type of medicine can be placed. So, each drawer can contain multiple pills (one per slot) but all pills must be of the same drug type.

Each Pill-Box drawer has a switch and a LED. The former is used to notify whether the pill in the drawer has been taken, and the latter is used to signal relevant information to the user. When the LED is *OFF*, it is not time to take the corresponding pill, while when the LED is *ON*, it means that the patient should assume that pill. When it is time to take a pill, the LED stays *ON* for 10 minutes after the scheduled time of the pill.

For each pill type, it is possible to set several deadlines throughout the day, meaning that the same drawer might be opened multiple times. However, if two or more pills have to be taken at the same time, the Pill-Box turns on only a single LED per time, by randomly choosing the order in which to assume them. Here we introduce three models for the ASMETA specification of the Pill-Box, where each one introduces new elements for refining time and pill management:

- *Ground model* (pillbox_ground): here we abstract the requirement that a drawer contains multiple slots and consider only a single pill per drawer. Moreover, time is not explicitly modeled, and information on the time passed is given by an external event (a monitored Boolean-valued function).
- *Model with time* (pillbox_time): this specification models time passing by a timer. We still keep the abstraction of having a single pill per drawer.
- *Final model* (pillbox_final): it captures all requirements of the Pill-Box system, and it thus specifies multiple pills (and multiple deadlines) per drawer.

These ASMETA models and all the other related artifacts are presented in part in this paper; their complete version can be found in *Models.zip* file at https://doi.org/10.5281/zenodo.12770854.

5 `AsmetaL`: The ASMETA Language

This section introduces the textual language `AsmetaL`, the user-facing language to define ASMETA models. The main modeling constructs of `AsmetaL` are here illustrated using the ground model pillbox_ground introduced in Sect. 4.

An ASMETA specification is described in a text file with extension `.asm` and structured as shown in Listing 1. It has five main sections:

```
asm pillbox_ground

import ../STDL/StandardLibrary
...
signature:
 // DOMAINS
 abstract domain Drawer
 enum domain LedLights = {OFF | ON}
 ...

 // FUNCTIONS
 dynamic monitored isPillTaken: Drawer −> Boolean
 ...
 dynamic controlled drawerLed: Drawer −> LedLights
 ...
 derived isOn: Drawer −> Boolean
 ...
 static drawer1: Drawer
```

```
definitions:
 // FUNCTIONS DEFINITIONS
 function isOn($d in Drawer) =
        (drawerLed($d) = ON)
 ...
 // RULE DEFINITIONS
 rule r_reset($drawer in Drawer) = ...
 ...
 // INVARIANTS AND PROPERTIES
 invariant inv_drawer1 over Drawer = ...
 ...
 // MAIN Rule
 main rule r_Main = ...
// INITIAL STATE
default init s0:
 // Turn−off all the LEDs for the Drawers
 function drawerLed($drawer in Drawer) = OFF
 ...
```

Listing 1. Structure of an ASMETA specification

- The section import allows us to include all or some of the declarations and definitions given in another ASMETA model.
- The section signature is where domains and functions are declared.
- The section definitions contains the definition of static concrete domains, static or derived functions, all transition rules, and possible state invariants, i.e., first-order formulas that must be true in all states.
- The section main rule defines the rule that is the starting point of the computation at each state; it may, in turn, call the other transition rules (defined as *macro call rules*[6]). A *run step* of an ASMETA model is the execution of all transition rules, which are directly or indirectly called from the main rule and are enabled to fire.
- The section default init introduces the initial values for dynamic concrete domains and dynamic functions declared in the signature.

Here we provide a more detailed look at each part of an ASMETA specification.

Specification Name. The first line of the specification contains the keyword asm followed by the name of the specification, which must be the same as the file. For instance:

```
asm pillbox_ground
```

indicates that the specification name is pillbox_ground and it must be defined in the file `pillbox_ground.asm`.

A model without the main rule is called a module[7]. It consists of declarations and definitions of domains, functions, invariants, macro call rules, and it can be imported by other ASMETA models. Note that an ASMETA model (the model that starts with the keyword asm) can be imported as well, except for the initial state and the main rule.

[6] Note that to define a macro call rule in the definitions section we use the syntax macro r_rule(*params*), while the macro rule is invoked from another rule as r_rule[*params*].

[7] A module name corresponds to the first word used in the .asm file.

Import. An AsmetaL specification can import modules, by using the file name with its relative or absolute path. For instance, the following line imports the StandardLibrary:

```
import ../STDL/StandardLibrary
```

The StandardLibrary is a user-ready module that defines names for basic domains and functions. This library is mandatory to import since it includes predefined names for primitive domains (like Boolean, Natural, Integer, etc.) and functions for the main operations over these domains and structured domains (for tuples, sequences, sets, bags, and maps). Other libraries are available[8] as explained in the following sections.

Signature Domains. The AsmetaL language allows the user to specify domains of different type:

- *Basic domains*: represent primitive data values and are denoted by ready-to-use domain symbols of the standard library (Boolean, Natural, Integer, Complex, Char, and String).
- *Enum domains*: finite enumeration of elements defined by the user.
- *Abstract domains*: (non-enumerable) user-defined domain to describe abstract entities of the real word.
- *Concrete domains*: user-named domain defined as sub-domain of another domain.
- *Structured domains*: representing structured data (like finite sets, tuples, maps) over other domains; examples are the Cartesian Product of two or more domains, and the mathematical Powerset of a domain.

Examples of user-defined domains from the ground model of the Pill-Box are:

```
abstract domain Drawer
enum domain LedLights = {OFF | ON }
enum domain Drugs = {TYLENOL | ASPIRINE | MOMENT}
```

Drawer is an abstract domain representing the drawer objects; such objects typically do not have a precise structure and the user further characterizes them by introducing functions over them (see next paragraph). LedLights is the enumeration for the light status of the LEDs; Drugs is the enumeration of three different types of drugs (Tylenol, Aspirine, and Moment).

Signature Functions. Basic functions form the basic signature of the machine and are classified into static, which never change during any run of the machine, and dynamic, that may be changed by the environment or by the machine updates. Dynamic functions are further divided into monitored, controlled, shared,

[8] https://github.com/asmeta/asmeta/blob/master/asm_examples/STDL/.

and out. AsmetaL adopts appropriate keywords for declaring all these kinds of functions. Examples of declarations of static functions are the constants[9] representing the three drawers (elements of the abstract domain Drawer):

```
static drawer1: Drawer
static drawer2: Drawer
static drawer3: Drawer
```

Examples of dynamic functions declaration in the ground model are[10]:

```
dynamic monitored isPillTaken: Drawer −> Boolean
dynamic monitored pillDeadlineHit: Drawer −> Boolean
dynamic controlled drawerLed: Drawer −> LedLights
dynamic controlled drug: Drawer −> Drugs
dynamic controlled isPillTobeTaken: Drawer −> Boolean
```

The function isPillTaken is a monitored function, and it is true when the user confirms he/she has taken the pill. Similarly, the monitored function pillDeadlineHit signals that the deadline for the pill contained in a specific drawer has come. A drawer contains a drug and has a drawerLed which is ON when it is time to take the pill. In addition, the Pill-Box uses the characteristic or indicator function isPillTobeTaken to store the drawers for which the deadline has been hit. These functions are controlled since their value is set by the machine.

In addition to basic functions, the modeler can introduce *derived* functions, i.e., those coming with a specification or computation mechanism defined in terms of other basic functions. Examples of declarations of derived functions are as follows:

```
derived isOn: Drawer −> Boolean
derived isOff: Drawer −> Boolean
derived areOthersOn: Drawer −> Boolean
```

These functions are used as guards in the transition rules and are defined (see below) in terms of the drawerLed controlled function.

Definitions Functions. Once declared, static and derived functions must also be defined explicitly in the definitions section. The notation to define functions is as in the following examples:

```
function isOn($d in Drawer) = (drawerLed($d) = ON)
function isOff($d in Drawer) = (drawerLed($d) = OFF)
function areOthersOn($d in Drawer) = switch($d)
   case drawer1 : isOn(drawer2) or isOn(drawer3)
   case drawer2 : isOn(drawer1) or isOn(drawer3)
   case drawer3 : isOn(drawer2) or isOn(drawer1)
endswitch
```

[9] The domain is optional. Functions of arity 0 are common *variables* of programming; 0-ary static functions are constants.

[10] The keyword **dynamic** is optional.

The right-hand term specifies the function law. In the case of the derived function areOthersOn, the right-end term is a logical map that associates domain elements to codomain elements. The target domains of the formal parameters are to be the same as those specified in the function declaration, and the domain type of the right-end term must be compatible with the function codomain. Note that, as exception to this explanation, static 0-ary functions (constants) over an abstract domain (such as drawer1) do not need to be defined.

Definitions Rules. An update rule is the basic form of a transition rule. Typically, an ASM transition system appears as a set of *guarded updates* or *conditional rules* of form **if** *cond* **then** *updates*, where function *updates* are simultaneously executed when the condition *cond* (also called "guard") evaluates to true. An example of a conditional rule is as follows:

```
if pillDeadlineHit($drawer) then isPillTobeTaken($drawer) := true endif
```

It sets to true the value of the function isPillTobeTaken for a given drawer $drawer[11] when it is time to take the drug of that drawer (denoted by the monitored function pillDeadlineHit).

In AsmetaL, the transition rules can be defined after the definition of concrete domains (if any) and functions. A rule definition starts with the keywords macro (it is optional) and rule, followed by the name of the rule with the fixed prefix r_, the list of free variables and their typing domains, and the rule body (containing occurrences of the free variables). As an example of rule definition, consider the rule r_reset that uses a par rule to reset the status of a given drawer (in parallel it sets the led to OFF and isPillTobeTaken to false):

```
rule r_reset($drawer in Drawer) = par
    drawerLed($drawer) := OFF
    isPillTobeTaken($drawer):= false endpar
```

Once defined, a named rule can be invoked (like in structured programming) within the rule body of another rule by using the rule name followed by the list of actual arguments (if any)[12] surrounded within square brackets (e.g., r_reset[$drawer]). When the rule is invoked, it is expanded by replacing every variable freely occurring within the rule body with the actual argument of the invocation (the association is positional).

The par rule and the forall rule are rule constructors realizing *synchronous parallelism* since both allow the synchronous parallel execution of multiple transition rules. The only difference is that the par rule expresses *bounded parallelism*, while the forall rule expresses potentially *unbounded parallelism*. An example of a forall rule in the Pill-Box ground model is in the rule definition:

[11] In AsmetaL the name of a variable freely occurring in a rule starts with the prefix $.

[12] The number of actual parameters must be equal to the number of the formal parameters of the rule to invoke and be domain-compatible with them. Invocations of rules of arity 0 is also allowed; in this case the list of parameters is empty.

```
rule r_setOtherDrawers = forall $drawer in Drawer do par
  if pillDeadlineHit($drawer) and isOff($drawer) then isPillTobeTaken($drawer) := true endif
  if isOn($drawer) and isPillTaken($drawer) then r_reset[$drawer] endif endpar
```

Such a rule, in parallel for all potential drawers, sets the status of a drawer if it is time to take the drawer's pill (pillDeadlineHit is true), or resets it in case the drawer's LED is on and the drawer's pill has already been taken. ASMETA supports non-deterministic operations, which are implemented by selecting a domain and picking a random element from it. This concept is realized by means of the choose rule. An example of rule definition that uses the choose rule is for the non-deterministic choice of one pill to take when there are more to take at a certain time.

```
rule r_choosePillToTake = choose $drawer in Drawer with
   isPillTobeTaken($drawer) and isOff($drawer) and not areOthersOn($drawer) do drawerLed($drawer) := ON
```

Since only a single red LED is to be on at a time, at each step the Pill-Box chooses randomly one still off among those of the drawers containing a pill to be taken, but only if all the other drawers' LEDs are off, and turns it (if any) on.

Definitions Invariants. Invariants allow users to specify first-order logic formulas that must be true in each computational state during model execution. In `AsmetaL`, invariants are defined after rule definitions but precede the main rule definition (see Sect. 6 for further details).

Definitions Properties. After the invariants, Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) properties can be defined in an ASMETA model (see Sect. 9 for further details).

Main Rule. The main rule designates the initial transition rule to execute (the entry point of the machine's program) at each computational step. Its definition follows those of invariants and properties, and has no formal parameters (its arity is 0). The main rule for the Pill-Box ground model is:

```
main rule r_Main = par
  r_choosePillToTake[]
  r_setOtherDrawers[] endpar
```

It is in charge of simultaneously (i) choosing one drawer with a pill to take (if any) and (ii) managing the state of the other drawers.

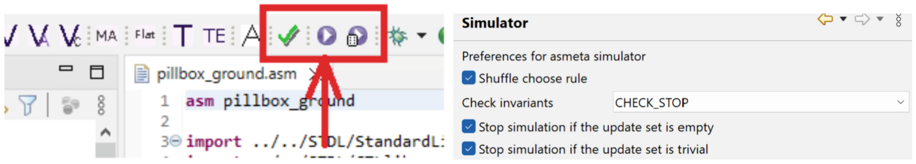
Initial State. An ASMETA specification may contain the initialization of controlled functions to the value that they must assume when the execution of the model starts. The syntax and rules to assign an initial value to a controlled function is the same for defining static/derived functions. For instance, in the following model fragment, the drawerLed function, for all drawers, is set to OFF, as well as the isPillTobeTaken function, which is set to false. Finally, the drug function associates a different type of drug to each drawer.

```
function drawerLed($drawer in Drawer) = OFF
function isPillTobeTaken($drawer in Drawer) = false
function drug($drawer in Drawer) = switch($drawer)
        case drawer1 : TYLENOL
        case drawer2 : ASPIRINE
        case drawer3 : MOMENT
    endswitch
```

If a function is not initialized, all its locations take the special value `undef`[13].

6 Model Simulation

Simulation is the first validation activity usually performed to check an ASMETA model's behavior during its development, and it is supported by the `AsmetaS` tool [5]. Given a model, at every step, the simulator builds the update set according to the theoretical definitions given in [15] to construct the model run. The simulator supports two types of simulation: *random* and *interactive*. In random mode, the simulator automatically assigns values to monitored functions, choosing them from their codomains. In interactive mode, instead, the user inserts the value of monitored functions and, in case of input errors, a message is shown inviting the user to insert again the function value. `AsmetaS` can be executed from the command line[14] and from the Eclipse interface. By using the Eclipse UI, the `AsmetaS` toolbar has three buttons (see Fig. 3) with three actions:



```
asm pillbox_ground

import ../../STDL/StandardL
```

Simulator

Preferences for asmeta simulator

☑ Shuffle choose rule

Check invariants CHECK_STOP

☑ Stop simulation if the update set is empty

☑ Stop simulation if the update set is trivial

Fig. 3. `AsmetaS` commands and options panel

✓ Parse the model and type check it

▶ Execute the model in interactive mode

▶ Execute the model with random inputs

[13] Although the parser does not force you to initialize all the controlled functions, it is strongly suggested to avoid run-time errors due to a missing initialization.

[14] More details are available in the Appendices file at https://doi.org/10.5281/zenodo.12770854.

```
 1  Running interactively pillbox_ground.asm
 2  INITIAL STATE:Drawer={drawer1,drawer2,drawer3}
 3  Insert a boolean constant for pillDeadlineHit(drawer1):
 4  true
 5  Insert a boolean constant for pillDeadlineHit(drawer2):
 6  false
 7  Insert a boolean constant for pillDeadlineHit(drawer3):
 8  false
 9  <State 0 (monitored)>
10  pillDeadlineHit(drawer1)=true
11  pillDeadlineHit(drawer2)=false
12  pillDeadlineHit(drawer3)=false
13  </State 0 (monitored)>
14  <UpdateSet − 0>
15  isPillTobeTaken(drawer1)=true
16  </UpdateSet>
17  <State 1 (controlled)>
18  drawerLed(drawer1)=OFF
19  drawerLed(drawer2)=OFF
20  drawerLed(drawer3)=OFF
21  isPillTobeTaken(drawer1)=true
22  isPillTobeTaken(drawer2)=false
23  isPillTobeTaken(drawer3)=false
24  </State 1 (controlled)>
25  Insert a boolean constant for isPillTaken(drawer1): false
26  <State 1 (monitored)>
27  isPillTaken(drawer1)=false
28  </State 1 (monitored)>
29  <UpdateSet − 1>
30  drawerLed(drawer1)=ON
31  </UpdateSet>
32  <State 2 (controlled)>
33  drawerLed(drawer1)=ON
34  drawerLed(drawer2)=OFF
35  drawerLed(drawer3)=OFF
36  isPillTobeTaken(drawer1)=true
37  isPillTobeTaken(drawer2)=false
38  isPillTobeTaken(drawer3)=false
39  </State 2 (controlled)>
40  Insert a boolean constant for isPillTaken(drawer1):
41  true
42  <State 2 (monitored)>
43  isPillTaken(drawer1)=true
44  </State 2 (monitored)>
45  <UpdateSet − 2>
46  drawerLed(drawer1)=OFF
47  isPillTobeTaken(drawer1)=false
48  </UpdateSet>
49  <State 3 (controlled)>
50  drawerLed(drawer1)=OFF
51  drawerLed(drawer2)=OFF
52  drawerLed(drawer3)=OFF
53  isPillTobeTaken(drawer1)=false
54  isPillTobeTaken(drawer2)=false
55  isPillTobeTaken(drawer3)=false
56  </State 3 (controlled)>
```

Fig. 4. Output of the interactive Simulation of the Pill-Box using `AsmetaS`

In the simulator option panel (see Fig. 3), the user can set the preferences regarding the choose rule, when to stop the random simulation (until the update set becomes empty or trivial), and how to handle invariants and axioms.

In Fig. 4, we show the result of the interactive simulation for the Pill-Box when the pill in drawer 1 hits the deadline (in State 0 - line 10), so the pill becomes to be taken (State 1 - line 21), the led becomes ON (State 2 - line 33), the user takes the pill, and the led becomes OFF (State 3 - line 50). Note that the update set is computed in the current state and is applied only in the next one. For instance, when the monitored location pillDeadlineHit(drawer1) is set true by the user in the initial state:

```
State 0 (monitored): pillDeadlineHit(drawer1)=true
```

the following rule:

```
if pillDeadlineHit($d) and isOff($d) then isPillTobeTaken($d) := true endif
```

checks the current state (State 0) and since the deadline is hit and the LED is off, the update set will contain the update of the location isPillTobeTaken(drawer1), which is updated only in the next state (State 1):

```
<UpdateSet − 0>
isPillTobeTaken(drawer1)=true
</UpdateSet>
<State 1 (controlled)>
isPillTobeTaken(drawer1)=true
...
</State 1 (controlled)>
```

Invariant Checking. `AsmetaS` implements an invariant checker, which (optionally) checks in every state reached during the computation if the invariants (if any) declared in the specification are satisfied or not. If an invariant is not satisfied, `AsmetaS` throws an `InvalidInvariantException`, which keeps track of the violated invariants and of the update set which has caused such violation. The invariant checker is particularly useful during the first phase of the model development to validate the specification. The designer adds model invariants, activates the invariant checker from the simulator options, and runs the model with some critical inputs. For example, with the following invariant:

```
invariant inv_drawer1 over Drawer: (forall $d in Drawer with isOff($d))
```

As soon as a led becomes ON, the computation stops:

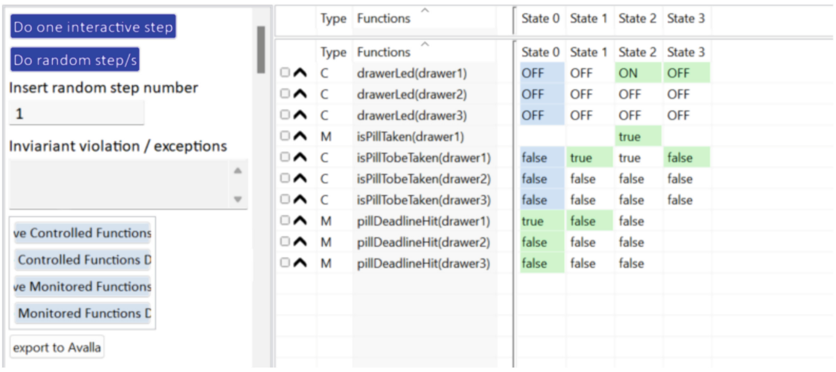
```
<State 2 (controlled)>
drawerLed(drawer2)=ON
...
</State 2 (controlled)>
INVARIANT violations
FINAL STATE: ....
run terminated
```

Consistent Updates Checking. `AsmetaS` is able to reveal inconsistent updates by throwing an UpdateClashException. The UpdateClashException records the location being inconsistently updated and the two different values assigned to it. The user, analyzing this error, can detect the fault in the specification. As the invariant checker, this feature is useful for model validation. For example, suppose to modify the r_setOtherDrawers rule by removing the strikethrough condition in the first conditional term as shown in the following code:

```
rule r_setOtherDrawers = forall $drawer in Drawer do par
    if pillDeadlineHit($drawer) ~~and isOff($drawer)~~ then isPillTobeTaken($drawer) := true endif
    if isOn($drawer) and isPillTaken($drawer) then r_reset[$drawer] endif endpar
```

The simulator signals an inconsistent update on the isPillTobeTaken(drawer1) location with the following message:

```
INCONSISTENT UPDATE FOUND !!! : location isPillTobeTaken(drawer1) updated to true != false
```



Insert random step number

1

Type	Functions	State 0	State 1	State 2	State 3
C	drawerLed(drawer1)	OFF	OFF	ON	OFF
C	drawerLed(drawer2)	OFF	OFF	OFF	OFF
C	drawerLed(drawer3)	OFF	OFF	OFF	OFF
M	isPillTaken(drawer1)			true	
C	isPillTobeTaken(drawer1)	false	true	true	false
C	isPillTobeTaken(drawer2)	false	false	false	false
C	isPillTobeTaken(drawer3)	false	false	false	false
M	pillDeadlineHit(drawer1)	true	false	false	
M	pillDeadlineHit(drawer2)	false	false	false	
M	pillDeadlineHit(drawer3)	false	false	false	

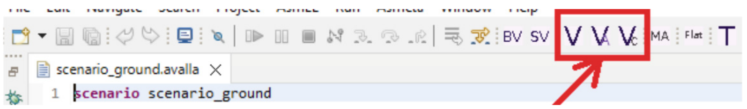
Fig. 5. `AsmetaA` Animation of the Pill-Box

Indeed, if the isOff($drawer) condition is removed and the deadline of the pill in the first drawer has passed, the first conditional rule sets isPillTobeTaken(drawer1) to true for all the following execution steps. However, if the LED for the drawer is ON and the user signals that the pill has been taken, the rule r_reset is executed and the location isPillTobeTaken(drawer1) is set to false. Thus, in the same step, the location is updated to two different values, leading to an inconsistent update.

6.1 Model Animation

The main disadvantage of the simulator is that it is textual, and this sometimes makes it difficult to follow the computation of the model. For this reason, ASMETA has a model animator, `AsmetaA` [11], which provides the user with complete information about all state locations, and uses colors, tables, and figures over simple text to convey information about states and their evolution. The animator helps the user follow the model computation and understand how the model state changes at every step. A screenshot of `AsmetaA` is shown in Fig. 5. To execute the animator, the user clicks on the A icon in Eclipse.

Similarly to the simulator, the animator supports *random* and *interactive* animation. In the interactive animation, the insertion of input functions is achieved through different dialog boxes depending on the type of function to be inserted (e.g., in the case of a Boolean function, the box has two buttons: one if the value is true and one if the value is false). If the function value is not in its codomain, the animator keeps asking until an accepted value is inserted. In random animation, the monitored function values are automatically assigned. With complex models, running one random step each time is tedious; for this reason, the user can also specify the number of steps to be performed and the tool performs the random simulation accordingly. In the case of invariant violation, a message is shown in a dedicated text box and the animation is interrupted (as it also happens in case of inconsistent updates). Once the user has animated the model, the tool allows exporting the model run as a scenario (see Sect. 7), so that it can be re-executed whenever desired. Figure 5 shows the animation of the Pill-Box



```
scenario scenario_ground
```

Fig. 6. `AsmetaV` commands

model using the same input sequence of the simulator. The result is the same, but the tabular view makes it easier to follow the state evolution.

7 Scenario-Based Validation

The `AsmetaS` and `AsmetaA` tools presented in the previous section require that the user executes the ASMETA model step by step or, at least, inserts some value to start the model simulation. In this section, we present the `AsmetaV` tool, which allows for performing scenario-based validation. Each *scenario* is a description of external actor actions and reactions of the system [18], which can be used to check the correct behavior of the model. Scenarios can be launched by using the button `V` shown in Fig. 6. Additionally, if the button $\mathtt{V_C}$ is pressed, `AsmetaV` keeps track of the rules covered by the scenario.

Scenarios are written in the `Avalla` language, and saved as `.avalla` files, as for the example reported in Listing 2 for the ASMETA ground model of the Pill-Box reported in Listing 1. The scenario models a simple assumption cycle for the pill in the first drawer. Initially, the Pill-Box has all the LEDs OFF, so no pill has to be taken (line 8-10). In the second step, we set the deadline for the pill in the first drawer as hit (line 17) and, after the execution of a step, the scenario checks whether the pill has been marked as one of those to be taken (line 20). Then, after a new execution step, we check that the LED corresponding to the first drawer is ON (line 25). Finally, after the patient has taken the pill, the scenario verifies whether all the LEDs have been turned OFF (line 35-37).

```
 1  scenario scenario_ground
 2  load pillbox_ground.asm
 3
 4  // Initially all deadlines are not hit
 5  set pillDeadlineHit(drawer1) := false;
 6  set pillDeadlineHit(drawer2) := false;
 7  set pillDeadlineHit(drawer3) := false;
 8  set isPillTaken(drawer3) := false;
 9  set isPillTaken(drawer1) := false;
10  set isPillTaken(drawer2) := false;
11  step
12  // Check that all leds are off
13  check drawerLed(drawer1) = OFF;
14  check drawerLed(drawer2) = OFF;
15  check drawerLed(drawer3) = OFF;
16  // Now, the time for the pill in the drawer 1 comes
17  set pillDeadlineHit(drawer1) := true;
18  step
19  // Check that pill is ready to be taken
20  check isPillTobeTaken(drawer1) = true;
21  check isPillTobeTaken(drawer2) = false;
22  check isPillTobeTaken(drawer3) = false;
23  step
24  // Check that the led for the drawer 1 is on
25  check drawerLed(drawer1) = ON;
26  check drawerLed(drawer2) = OFF;
27  check drawerLed(drawer3) = OFF;
28  check isPillTobeTaken(drawer1) = true;
29  check isPillTobeTaken(drawer2) = false;
30  check isPillTobeTaken(drawer3) = false;
31  // Now, take the pill
32  set isPillTaken(drawer1) := true;
33  step
34  // Check that the led is reset
35  check drawerLed(drawer1) = OFF;
36  check drawerLed(drawer2) = OFF;
37  check drawerLed(drawer3) = OFF;
38  check isPillTobeTaken(drawer1) = false;
39  check isPillTobeTaken(drawer2) = false;
40  check isPillTobeTaken(drawer3) = false;
```

Listing 2. Example of `Avalla` scenario

Scenario Name. The first line of the scenario defines its name. For instance:

```
scenario scenario_ground
```

Unlike the ASMETA specification, the scenario name is not required to match the file name.

Loading AsmetaL Specifications. Each Avalla scenario is executed against an ASMETA spec. Thus, after having defined the scenario name it is essential to specify which ASMETA model to load. This is done by using the load command, followed by the relative or absolute path of the .asm file (including its extension):

```
load pillbox_ground.asm
```

Setting Monitored Functions. Monitored functions are read by the machine from the environment. When performing scenario-based validation, the user may supply the values for monitored or shared functions through the set command. These functions are then used as input signals to the system. For instance:

```
set pillDeadlineHit(drawer1) := false;
```

is used to set the monitored function pillDeadlineHit for the drawer1 to false.

Step Execution. After having set the value for the monitored functions of interest, an ASMETA computation step (i.e., the reaction of the system) can be launched by using the step command. Additionally, Avalla supports the execution of multiple steps using the stepUntil command, until a specified Boolean condition becomes true.

Checking Controlled Functions. Executing an ASMETA specification step will lead to the update of the internal state of the ASMETA model. The check command is used to inspect property values in the current state of the underlying model. For instance:

```
check drawerLed(drawer1) = OFF;
```

checks that the controlled function drawerLed for the drawer1 is OFF. When executing an Avalla scenario, the AsmetaV validator captures any check violation, and, if none occurs, it finishes with a "PASS" verdict ("FAIL" otherwise).

AsmetaL Code in Avalla Scenarios. Avalla scenarios support basic set commands. However, users may want to set ASMETA functions by using a more complex set of instructions, e.g., rules previously defined in the ASMETA specification or by parallelizing the update. Thus, scenarios allow for including AsmetaL commands with the exec keyword. For instance, the following Avalla code

```
set pillDeadlineHit(drawer1) := false;
set pillDeadlineHit(drawer2) := false;
set pillDeadlineHit(drawer3) := false;
```

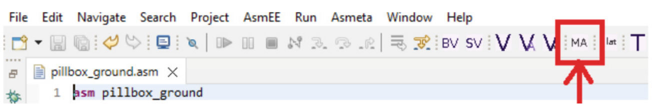
can be replaced by

```
exec forall $drawer in Drawer do pillDeadlineHit($drawer) := false;
```

Note that this command would have been wrong if written in an AsmetaL specification, as pillDeadlineHit is a monitored function and it should not be set by the system. However, when AsmetaV simulates the scenario, a new ASMETA spec is created, and the monitored functions are converted to controlled ones, whose value is set to that specified in the Avalla scenario (either with a set command or with the exec command).

Scenario Modularization. The user can exploit modularization also during scenario building. Indeed, it is possible to define blocks, i.e., sequences of set, step, and check, that can be recalled using the execblock command when writing other scenarios that foresee the same sequence of Avalla commands.

Exporting and Animating Scenarios. Avalla scenarios can be exported from the AsmetaA tool, so that an animation session can automatically be repeated multiple times (see the "export to Avalla" button in Fig. 5). Similarly, AsmetaV supports the execution of scenarios through animation, by using the button $\mathtt{V_A}$ shown in Fig. 6. This allows users to control execution, enabling step-by-step scenario execution.

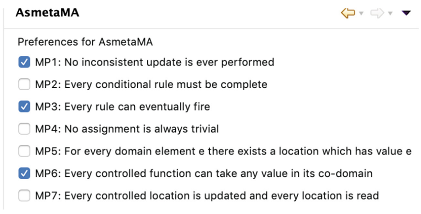


```
asm pillbox_ground
```

Fig. 7. AsmetaMA command

8 Model Review

ASMETA supports a form of static analysis of a model to automatically capture typical modeling errors such as inconsistent updates or dead specification parts



AsmetaMA

Preferences for AsmetaMA

- [✓] MP1: No inconsistent update is ever performed
- [ ] MP2: Every conditional rule must be complete
- [✓] MP3: Every rule can eventually fire
- [ ] MP4: No assignment is always trivial
- [ ] MP5: For every domain element e there exists a location which has value e
- [✓] MP6: Every controlled function can take any value in its co-domain
- [ ] MP7: Every controlled location is updated and every location is read

(a) Meta-properties

```
MP1: No inconsistent update is ever performed
Location isPillTobeTaken(DRAWER1) is updated
to values TRUE and FALSE when are satisfied
simultaneously the conditions
(TRUE & pillDeadlineHit(DRAWER1)) and
(TRUE & (isOn(DRAWER1) & isPillTaken(DRAWER1)))
```

(b) Violation of MP1

Fig. 8. AsmetaMA usage

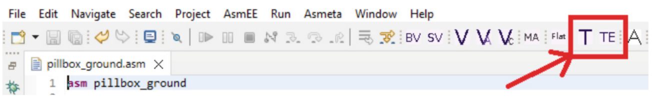


Fig. 9. AsmetaSMV command

(transition rules that are never triggered) due to overspecification. We called such a kind of static analysis about model quality *automatic model review* and it is carried out by the AsmetaMA tool [3], which can be executed by clicking on the button shown in Fig. 7. This tool checks the presence of seven types of errors by using suitable *meta-properties* specified in CTL and verified using the model checker AsmetaSMV (see Sect. 9). Figure 8a shows the selection of the seven meta-properties in AsmetaMA. An example of meta-property is MP1, which checks the presence of inconsistent updates. Figure 8b reports an example of inconsistent update revealed by AsmetaMA on the same example reported in Sect. 6.

9 Formal Verification Through Model Checking

Besides validation, the ASMETA toolset supports the user in the properties' verification activity by the tool AsmetaSMV [2]. Properties are written in terms of propositional formulas over the machine's signature, preceded by the keyword ctlspec or ltlspec. For this purpose the libraries CTLLibrary.asm and LTLLibrary.asm must be imported, so for each CTL/LTL operator an equivalent AsmetaL Boolean-valued function is defined. The following example shows a CTL property (with the temporal operator *AG Φ* - globally *Φ*) for the Pill-Box ground model, i.e. a propositional formula that must hold in all reachable states:

```
ctlspec ag((forall $d in Drawer with isOn($d) implies (not areOthersOn($d))))
```

These properties are then automatically translated into a model of the symbolic model checker NuSMV [20], used to perform the verification. If the ASMETA model contains infinite or time domains, the NuXmv [19] model checker is preferred. The choice of the model checker is performed in Eclipse from the ASMETA → AsmetaSMV preferences. The buttons shown in Fig. 9 are used to verify the specification: the first button translates the specification into a model for the model checker without executing it, and the second translates and executes the specification using the selected model checker. The output of the model checker is pretty printed in terms of elements of the ASMETA signature. If the property is positively verified, the `AsmetaSMV` tool prints out on the Eclipse console that the property is true:

```
-- specification AG (((drawerLed(DRAWER1) = ON -> !areOthersOn(DRAWER1)) &
(drawerLed(DRAWER2) = ON -> !areOthersOn(DRAWER2))) & (drawerLed(DRAWER3) = ON
-> !areOthersOn(DRAWER3))) is true
```

Otherwise, assuming the property is false, it returns a counterexample. If we want to verify that a pill in drawer1 is always taken when the pill deadline hits, we can write the following property:

```
ctlspec ag(pillDeadlineHit(drawer1) implies af(isOn(drawer1)))
```

When running the model checker, the property is false because it can happen that the pill in drawer1 will never be taken (the function isPillTaken(drawer1) is never set to true), and the counterexample in Listings 3 is printed.

```
-- specification AG (pillDeadlineHit(DRAWER1) ->
AF drawerLed(DRAWER1) = ON) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    pillDeadlineHit(DRAWER1) = false
    drawerLed(DRAWER1) = OFF
    drawerLed(DRAWER2) = OFF
    isPillTobeTaken(DRAWER2) = false
    drawerLed(DRAWER3) = OFF
    isPillTobeTaken(DRAWER3) = false
    ...
  -> State: 1.2 <-
    pillDeadlineHit(DRAWER1) = true
    pillDeadlineHit(DRAWER2) = true
  -> State: 1.3 <-
    pillDeadlineHit(DRAWER1) = false
    isPillTobeTaken(DRAWER2) = true
    isPillTobeTaken(DRAWER1) = true
    pillDeadlineHit(DRAWER2) = false
  -> State: 1.4 <-
    ...
    pillDeadlineHit(DRAWER2) = true
    areOthersOn(DRAWER3) = true
    areOthersOn(DRAWER1) = true
  -> State: 1.5 <-
    drawerLed(DRAWER2) = OFF
    isPillTaken(DRAWER2) = false
    pillDeadlineHit(DRAWER2) = false
    areOthersOn(DRAWER3) = false
    areOthersOn(DRAWER1) = false
```

Listing 3. Counterexample generated by `AsmetaSMV`

10 Model Refinement

After having performed the activities presented in the previous sections, the model can be refined and the desired level of detail can be achieved. Here we report details of the two model refinements introduced in Sect. 4, and we highlight the differences between the ground model and the refined models.

10.1 Time Handling: pillbox_time

Modeling. The first model refinement we propose consists in explicitly modeling time passing, which is left abstract in the pillbox_ground model, by introducing the timer tenMinutes to capture the requirement stating that *the LED* *stays on for 10 min after the scheduled time to take the pill.* Dealing with timers requires importing the predefined time library and setting a suitable timer as an element of the abstract domain Timer:

```
import ../STDL/TimeLibrarySimple
static tenMinutes: Timer
```

The time library provides the user with several features to check whether *a timer is expired* (see the use of the predicate expired(tenMinutes) in rule r_take in the pillbox_time model to control the expiration of timer tenMinutes) and *to reset a timer* (see the use of the predefined rule r_reset_timer[tenMinutes] in rule r_pillToBeTaken of pillbox_time model). Using a timer always requires initializing the timer's duration and its starting time; in the pillbox_time model, the duration of the timer tenMinutes is set to 600 time unit (seconds, in this case) and its starting time is equal to the current time (e.g., taken as monitored value from the Java virtual machine).

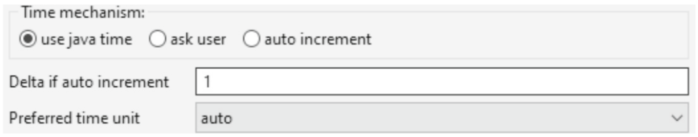
```
function duration($t in Timer) = 600 // Timer initialization
function start($t in Timer) = currentTime($t)
// From the Time library
function currentTime($t in Timer) = mCurrTimeSecs
```

This model is an example of *vertical* model refinement, where concepts or behaviors previously left abstract are modeled in detail. Here the monitored function pillDeadlineHit is refined by the homonymous derived function that relates the time of a pill consumption with the current time. The value of the function time_consumption is set, for each pill/drawer, in the initialization section of the pillbox_time model, as follows:

```
function time_consumption($drawer in Drawer) = switch($drawer) // Initialization of the time consumption
 case drawer1 : 60
 case drawer2 : 2400
 case drawer3 : 180
endswitch
```

The behavior of the rule r_choosePillToTake is refined by adding the new rule r_pillToBeTaken to turn on the led and reset the timer tenMinutes if the led is off. The behavior of rule r_setOtherDrawers is also refined by marking a pill to be taken if its time of consumption is reached and by resetting the timer (of a drawer with red led) if the pill has been taken or the timer of ten minutes waiting has expired.

Validation and Verification. As explained in the previous sections, V&V activities can be performed on this refinement level. Since this refinement step considers also the time during the simulation, the simulator (as well as the animator) handles the time using three different approaches for setting the monitored mCurrTimeSecs [10] (see Fig. 10): 1. time is read from the machine using the Java TimeAPI; 2. the user enters the value for time (like for monitored functions); 3.


Time mechanism: [✓ use java time] ask user auto increment

Delta if auto increment: 1

Preferred time unit: auto

Fig. 10. `AsmetaS` time simulation preferences

time is automatically increased at each step by a predefined value. Additionally, ASMETA allows the user to set the preferred time unit.

Regarding property verification, the NuSMV model checker does not support infinite domains (such as in the case of times), so the NuXmv [19] model checker must be used. However, its integration with ASMETA is still under development and not stable, thus, here we do not discuss its use.

10.2 Managing Multiple Pills: pillbox_final

Modeling. A further (and the last that we propose) vertical model refinement specifies the complete Pill-Box functionalities, allowing modeling the requirement that *Each drawer contains multiple slots (one for each pill) that are emptied in sequence.* To model this requirement we introduce the following controlled functions time_consumption and drugIndex:

```
dynamic controlled time_consumption: Drawer −> Seq(Integer)
dynamic controlled drugIndex: Drawer −> Natural
```

The former maps each drawer into a sequence of integers, containing the time deadlines expressed in seconds. The latter associates with each drawer an integer indicating the next slot to be emptied in the corresponding drawer. The two functions are initialized accordingly:

```
function time_consumption($drawer in Drawer) = switch($drawer) // Initialization of the time consumption
 case drawer1 : [60, 1200, 1800]
 case drawer2 : [2400, 3000, 3600]
 case drawer3 : [180, 1200, 1800]
endswitch
function drugIndex($drawer in Drawer) = 0n // Every drawer has an index starting from 0
```

The derived function pillDeadlineHit is refined to check the pill's deadline in the drawer's current slot to be emptied[15]. The newly derived function isThereAnyOtherDeadline indicates if there is any other pill in the drawer to be taken. This information is used to refine the rule r_setOtherDrawers, which leads to suitably updating the drawer state (led status and drug index) by invoking the (nested and refined) macro call rule r_reset.

[15] The function at(sequence,i) yield the value of the ith element of the sequence.

Listing 4. Header file

```
#define ANY String
#include <string.h>
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <list>
#include <chrono>
#include "../../STDL/TimeLibrarySimple.h"
using namespace std;
/* DOMAIN DEFINITIONS */
namespace pillbox_finalnamespace{
    class Drawer;
    enum LedLights {OFF, ON};
    enum Drugs {TYLENOL, ASPIRINE, MOMENT};
}
using namespace pillbox_finalnamespace;
class pillbox_finalnamespace::Drawer{
public:
    static set<Drawer*> elems;
    Drawer(){elems.insert(this);}
};
class pillbox_final : public virtual TimeLibrarySimple{
    /* DOMAIN CONTAINERS */
    const set<LedLights> LedLights_elems;
    const set<Drugs> Drugs_elems;
public:
    /* FUNCTIONS */
    map<Drawer*, bool> isPillTaken;
    map<Drawer*, LedLights> drawerLed[2];
    map<Drawer*, vector<int>> time_consumption[2];
    ...
    static Timer* tenMinutes;
    bool isOn (Drawer* param0_isOn);
    bool isOff (Drawer* param0_isOff);
    ...
    static Drawer* drawer1;
    ...
    /* RULE DEFINITION */
    void r_reset (Drawer* _drawer);
    void r_pillToBeTaken (Drawer* _drawer);
    ...
    void r_Main();
    pillbox_final();
    void initControlledWithMonitored();
    void getInputs();
    void setOutputs();
    void fireUpdateSet();
};
```

Listing 5. Cpp file

```
#include "pillbox_final.h"
using namespace pillbox_finalnamespace;
/* Conversion of ASM rules in C++ methods */
void pillbox_final::r_reset (Drawer* _drawer){
    drawerLed[1][_drawer] = OFF;
    drugIndex[1][_drawer] = (drugIndex[0][_drawer] + 1);
    isPillTobeTaken[1][_drawer] = false;
}
void pillbox_final::r_pillToBeTaken (Drawer* _drawer){ ... }
void pillbox_final::r_ON (Drawer* _drawer){ ... }
void pillbox_final::r_choosePillToTake(){ ... }
void pillbox_final::r_setOtherDrawers(){ ... }
void pillbox_final::r_Main(){
    r_choosePillToTake();
    r_setOtherDrawers();
}
/* Static function definition */
bool pillbox_final::isOn(Drawer* _d){
    return (drawerLed[0][_d] == ON);
}
bool pillbox_final::isOff(Drawer* _d){ ... }
bool pillbox_final::areOthersOn(Drawer* _d){ ... }
bool pillbox_final::pillDeadlineHit(Drawer* _d){ ... }
bool pillbox_final::isThereAnyOtherDeadline(Drawer* _d){ ... }
/* Function and domain initialization */
pillbox_final::pillbox_final(): LedLights_elems({OFF,ON}),
    Drugs_elems({TYLENOL,ASPIRINE,MOMENT}) {
    /* Init static functions Abstract domain */
    tenMinutes = new Timer;
    ...
    /* Function initialization */
    for(const auto& _drawer : Drawer::elems){
        drawerLed[0].insert({_drawer,OFF});
        drawerLed[1].insert({_drawer,OFF});
    } ...
}
/* Apply the update set */
void pillbox_final::fireUpdateSet(){
    drawerLed[0] = drawerLed[1];
    time_consumption[0] = time_consumption[1];
    drug[0] = drug[1];
    drugIndex[0] = drugIndex[1];
    ...
}
/* init static functions and elements of abstract domains */
set< Drawer*> Drawer::elems;
Timer* pillbox_final::tenMinutes;
Drawer* pillbox_final::drawer1;
...
```

Remark. Model refinement must be proved to be *correct*, i.e., at each refinement step, a refined model must be proved to be a correct refinement of the abstract one. Due to lack of space and to keep this presentation easy to follow, here we skip the proof of correct refinement of models and the application of the `AsmRefProver` supporting automatic proof of a particular form of model refinement.

11 From an ASMETA Model to Code

As requested by the best practices of model-driven engineering [16], the implementation of a system should be obtained from its model through a systematic model-to-code transformation. ASMETA features a set of tools allowing the automatic generation of C++ code [12] and C++ unit tests, and Java code [9].

In the following, we focus on using the `Asmeta2C++` tool. It generates C++ code (which is meant to be integrated with other artifacts or directly embedded in the final device) starting from an ASMETA model and, in particular, it produces two files: header (.h) and source (.cpp). The former contains the interface of the source file and the translation of model domain declarations and definitions, function and rule declarations. The latter includes rules implementation, the functions and domain initialization, and the definitions of the functions. `Asmeta2C++` is only available as a command line tool and can be executed, in the case of the last refinement, with the following command:

```
java -jar Asmeta2Cpp.jar pillbox_final.asm
```

Additional options for the previous command are available in the Appendices file at https://doi.org/10.5281/zenodo.12770854.

An excerpt of the translation of the Pill-Box case study in C++ is shown in Listings 4 and 5, while the complete version of the source code is available in the replication package. An ASM run step involves executing the main rule and updating the locations. In C++, this is realized through two methods: mainRule() for translating the ASMETA main rule and fireUpdateSet() for updating locations to their next state values. `Asmeta2C++` can generate two additional files allowing to embedding the generated class into an Arduino program. Further insights into the translation of ASMETA rules and constructs into corresponding C++ instructions are given in [12].

12 Conclusion

This tutorial provides an overview of ASMETA, an integrated set of tools to describe the behavior of discrete event systems using the ASM formalism. The hands-on approach adopted in this tutorial shows how to combine all the model analysis techniques offered by ASMETA in order to start from a ground or partial specification of the system behavior, and then refine it incrementally into more complete models till leading to transformation to other external analysis models or code. Thanks to the adoption of a set of integrated and easy-to-use tools, like ASMETA, the effort for modeling and analysis with a formal method, like ASM, may be reduced and more software engineers may be convinced of applying the formal method for richer system design and more reliable systems.

Data Availability Statement. The artifacts for the tutorial paper are available at https://doi.org/10.5281/zenodo.12770854.

References

1. Arcaini, P., Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: The ASMETA Approach to Safety Assurance of Software Systems, pp. 215–238. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-76020-5_13
2. Arcaini, P., Gargantini, A., Riccobene, E.: AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) Abstract State Machines, Alloy, B and Z, pp. 61–74. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_6

3. Arcaini, P., Gargantini, A., Riccobene, E.: Automatic review of Abstract State Machines by meta property verification. In: Muñoz, C. (ed.) Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215, pp. 4–13. NASA, Langley Research Center, Hampton VA 23681–2199, USA (2010)
4. Arcaini, P., Gargantini, A., Riccobene, E.: SMT-based automatic proof of ASM model refinement. In: De Nicola, R., Kühn, E. (eds.) Software Engineering and Formal Methods, pp. 253–269. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_17
5. Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P.: A model-driven process for engineering a toolset for a formal method. Softw. Pract. Exper. **41**, 155–166 (2011). https://doi.org/10.1002/spe.1019
6. ter Beek, M.H.: Formal methods and tools applied in the railway domain. In: Bonfanti, S., Gargantini, A., Leuschel, M., Riccobene, E., Scandurra, P. (eds.) Rigorous State-Based Methods - 10th International Conference, ABZ 2024, Bergamo, Italy, June 25-28, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14759, pp. 3–21. Springer (2024). https://doi.org/10.1007/978-3-031-63790-2_1
7. ter Beek, M.H., et al.: Formal methods in industry. Form. Asp. Comput. (2024)
8. Bombarda, A., Bonfanti, S., Gargantini, A.: Developing medical devices from abstract state machines to embedded systems: a smart pill box case study. In: Mazzara, M., Bruel, J.M., Meyer, B., Petrenko, A. (eds.) Software Technology: Methods and Tools, pp. 89–103. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-29852-4_7
9. Bombarda, A., Bonfanti, S., Gargantini, A.: From concept to code: unveiling a tool for translating abstract state machines into java code. In: Rigorous State-Based Methods 10th International Conference, ABZ 2024, Bergamo, Italy, June 25-28, 2024, Proceedings, Lecture Notes in Computer Science, vol. 14759. Springer (2024). https://doi.org/10.1007/978-3-031-63790-2_10
10. Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E.: Extending ASMETA with time features. In: Raschke, A., Méry, D. (eds.) Rigorous State-Based Methods, pp. 105–111. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-77543-8_8
11. Bonfanti, S., Gargantini, A., Mashkoor, A.: ASMETAA: animator for abstract state machines. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z, pp. 369–373. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_25
12. Bonfanti, S., Gargantini, A., Mashkoor, A.: Design and validation of a C++ code generator from abstract state machines specifications. J. Softw.: Evol. Process **32**(2), e2205 (2020). https://doi.org/10.1002/smr.2205
13. Börger, E.: The ASM refinement method. Form. Asp. Comput. **15**, 237–257 (2003)
14. Börger, E., Raschke, A.: Modeling Companion for Software Practitioners. Springer, Berlin, Heidelberg (2018). https://doi.org/10.1007/978-3-662-56641-1
15. Börger, E., Stärk, R.: Abstract State Machines. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
16. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Springer International Publishing (2017). https://doi.org/10.1007/978-3-031-02549-5
17. Broy, M., et al.: Does every computer scientist need to know formal methods? Form. Asp. Comput. (2024). https://doi.org/10.1145/3670795
18. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: A scenario-based validation language for ASMs. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.)

Abstract State Machines, B and Z, pp. 71–84. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_7

19. Cavada, R., et al.: The NUXMV symbolic model checker. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification, pp. 334–342. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
20. Cimatti, A., et al: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification, pp. 359–364. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
21. Garavel, H., Beek, M.H.t., Pol, J.V.D.: The 2020 expert survey on formal methods. In: Formal Methods for Industrial Critical Systems: 25th International Conference, FMICS 2020, Vienna, Austria, September 2–3, 2020, Proceedings 25, pp. 3–69. Springer (2020). https://doi.org/10.1007/978-3-030-58298-2_1
22. Gleirscher, M., Marmsoler, D.: Formal methods in dependable systems engineering: a survey of professionals from Europe and North America. Empir. Softw. Eng. **25**(6), 4473–4546 (2020). https://doi.org/10.1007/s10664-020-09836-5
23. Gleirscher, M., van de Pol, J., Woodcock, J.: A manifesto for applicable formal methods. Softw. Syst. Model. **22**(6), 1737–1749 (2023). https://doi.org/10.1007/s10270-023-01124-2

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Practical Deductive Verification of OCaml Programs

Mário Pereira(✉)

NOVA LINCS, NOVA School of Science and Technology, Lisbon, Portugal
mjp.pereira@fct.unl.pt

Abstract. In this paper, we provide a comprehensive, hands-on tutorial on how to apply deductive verification to programs written in OCaml. In particular, we show how one can use the GOSPEL specification language and the Cameleer tool to conduct mostly-automated verification on OCaml code. In our presentation, we focus on two main classes of programs: first, purely functional programs with no mutable state; then on imperative programs, where one can mix mutable state with subtle control-flow primitives, such as locally-defined exceptions.

Keywords: Deductive Software Verification · OCaml · Cameleer · GOSPEL

1 Introduction

Deductive software verification [11] is a subject within the larger field of formal methods [23]. One can define deductive software verification as the process of expressing the correctness of a program as a mathematical statement, then proving it. However, such a definition does not properly highlight the connection between the three main components in deductive verification: the *logical specification*, which mathematically captures *what* one wishes to compute; the *code*, which stands for *how* one materializes ideas as a piece of software; and, finally, a formal proof of *why* the code adheres to the given specification. This last component can be realized via a so-called *Verification Conditions Generator*, a mechanical tool that takes as input the code and the specification, producing the aforementioned correctness statement.

In this tutorial paper, we focus on the deductive verification of programs developed in the OCaml language. We use the Cameleer [27] tool to verify, in a mostly-automated fashion, that an OCaml program adheres to its specification. One key aspect of our presentation is the use of GOSPEL [8], the *G*eneric *O*Caml

This work is partly supported by Agence Nationale de la Recherche (ANR) grant ANR-22-CE48-0013-01 (GOSPEL) and NOVA LINCS ref. UIDB/04516/2020 (https://doi.org/10.54499/UIDB/04516/2020) and ref. UIDP/04516/2020 (https://doi.org/10.54499/UIDP/04516/2020) with the financial support of FCT.IP.

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 518–542, 2025.
https://doi.org/10.1007/978-3-031-71177-0_29

*SPE*cification *L*anguage. This is a tool-agnostic language, which serves as a common ground for the different OCaml verification tools and techniques. One important feature of GOSPEL is that specifications are written in a subset of the OCaml language, plus quantifiers, making the adoption of formal methods even more appealing for the working OCaml programmer.

Throughout this tutorial, we harmoniously combine an algorithmic discovery journey with the art of deductive software verification. We believe verification tools and techniques are better presented through the lens of classical data structures and algorithms. We believe this hands-on, example-oriented approach is a more efficient and convincing way to justify the interest in using verification tools. In order to follow this tutorial, we only assume the reader to possess basic knowledge of functional programming (not necessarily in OCaml) and some knowledge of deductive verification, at least to the level of understanding function contracts, loop invariants, and proofs by induction.

This paper is organized as follows. Section 2 provides an overview of the GOSPEL language. Section 3 introduces the Cameleer tool, mainly using two examples of verified OCaml programs, the first being a pure implementation, the other featuring mutability. In Sect. 4, we take a more in-depth dive into the verification of functional programs. Section 5 extends our class of verified programs, incorporating some imperative traits of the OCaml language. We terminate with some related work (Sect. 6) and closing remarks and future perspectives (Sect. 7). All the software and proofs used in this paper are publicly available in a companion artifact [28], which also complements this paper with other case studies verified in Cameleer.

2 A Primer on GOSPEL

GOSPEL is a behavioral specification language for OCaml code. It is a contract-based, statically typed language, with a formal semantics defined by means of translation into Separation Logic [6,30]. The term *Generic* comes from the fact that GOSPEL is not tied to any particular tool or analysis technique. In fact, nowadays, one can use GOSPEL to attach specifications to OCaml that are then analyzed using *runtime assertion checking* techniques [13], or formally verified using deductive verification tools [7,27]. GOSPEL is inspired by other behavioral specification languages [15], such as JML [19] or Eiffel [22]. However, both JML and Eiffel require the specification to always be executable. This in not the case in GOSPEL. In this tutorial paper, we focus on the use of GOSPEL for deductive verification.

When compared to other specification languages based on Separation Logic, *e.g.*, VeriFast [17], Viper [24], or Gillian [21], GOSPEL takes a different design choice: permission and separation conditions are implicitly associated with function arguments, which greatly improves conciseness over Separation Logic. We argue this is an important argument in favor of GOSPEL adoption by regular OCaml programmers, who are not necessarily proof experts. We believe it is of crucial importance to develop the languages and tools that bring practitioners into formal methods.

```
type 'a t
(*@ mutable model view: 'a list *)

val create : unit -> 'a t
(*@ s = create ()
      ensures s.view = [] *)

val is_empty : 'a t -> bool
(*@ b = is_empty s
      ensures b <-> s.view = [] *)

val push : 'a -> 'a t -> unit
(*@ push x s
      modifies s.view
      ensures  s.view = x :: old s.view *)

val pop : 'a t -> 'a
(*@ v = pop s
      requires s.view <> []
      modifies s.view
      ensures  v :: s.view = old s.view *)
```

Fig. 1. GOSPEL-annotated Stack Interface.

GOSPEL was initially designed as an interface behavioral specification language. The interface shown in Fig. 1 exemplifies the use of GOSPEL to specify an OCaml interface for a *polymorphic stack* data structure, independent of the underlying implementation (it could be, *e.g.*, a linked-list, a ring buffer, etc.). GOSPEL specification is given within comments of the form `(*@ ... *)`. We start by specifying that type `t` of stacks is described, at the logical level, via a *model field* named `view`. This field is of type `'a list` (here, we use OCaml's immutable lists) and describes the sequence of elements contained in the data structure. There is, however, one important aspect about the use of `view`: it is declared as a `mutable` field, which means one should expect in-place modifications to the stack. In other words, `t` represents an imperative data structure.

When it comes to attaching specification to functions, the first line in the GOSPEL comments names function arguments and its return value. To describe the behavior of a function, we mainly use three clauses: `requires`, to introduce a precondition; `ensures`, to introduce a postcondition; and `modifies`, which enumerates all the mutable fields changed during the call to a function. For instance, functions `create` and `is_empty` are simply annotated with postconditions stating, respectively, that a fresh stack is created with no elements and, conversely, a stack is empty if it does not contain any element. Finally, functions `push`, `pop`, and `transfer` modify the contents of a stack via side-effects. The former inserts a new element to the top of the `view` model; the latter removes the top element, assuming as a precondition that the stack is not empty. The term `old s.view`

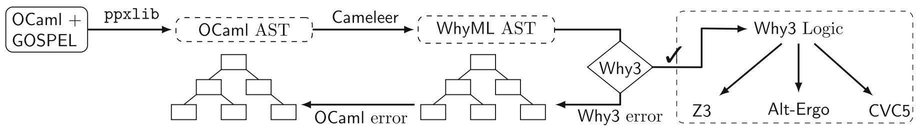


Fig. 2. Cameleer Architecture and verification pipeline, taken from [27].

represents the pre-state of model `view`, *i.e.*, the value of this field at the moment the function is called.

Throughout this tutorial, we will use and discover GOSPEL features via OCaml examples formally verified using the Cameleer tool. However, we are not able to cover here all the relevant aspects of the language in full detail. For a more in-depth presentation of GOSPEL, we refer the reader to the original paper [8] and the user manual[1].

3 The Cameleer Verification Tool

Up until recently, programmers would face a difficult choice if they wished to produce verified OCaml code: either conduct automated proof, but entirely re-implement their code-bases in a proof-aware language, and then rely on code extraction; or verify actual OCaml code, but using an interactive proof assistant, with the burden of manual proofs. Cameleer offers a compromise between the two approaches: it is a tool for the deductive verification of OCaml-written programs, with a clear focus on proof automation. It aims to provide an easy to use framework for the specification and verification of OCaml code.

Figure 2 presents the verification pipeline of the tool. It takes as input an OCaml implementation file, annotated using GOSPEL, and translates it into an equivalent WhyML program, the programming and specification language of the Why3 framework [14]. Why3 is a tool-set for the deductive verification of software, oriented towards automated proof. A distinctive feature of Why3 is that it can interface with different off-the-shelf SMT solvers, which greatly increases proof automation.

With Cameleer, we put forward the vision of the *specifying programmer*: those who write the code, should also be able to specify it. But, we want to push this vision even further: those who write the code, should be able to specify it *and formally verify it*. Leveraging on the proof automation and tool-set offered by Why3, we believe Cameleer is a good candidate to fill this need in the working OCaml programmers community.

In this section, we introduce Cameleer via two examples of mechanically verified algorithms implemented in OCaml and specified in GOSPEL. The first one is a traditional merge operation over sorted lists. The second one is a linear-search operation on arrays, hence an imperative implementation, featuring some interesting constructions from the OCaml language. The two examples have a

[1] https://ocaml-gospel.github.io/gospel/.

```
module type PRE_ORD = sig
  type t

  (*@ predicate le (x : t) (y : t) *)

  (*@ axiom reflexive : forall x. le x x *)
  (*@ axiom total     : forall x y. le x y \/ le y x *)
  (*@ axiom transitive: forall x y z. le x y -> le y z -> le x z *)

  val leq : t -> t -> bool
  (*@ b = leq x y
        ensures b <-> le x y *)
end
```

Fig. 3. Type Equipped With a Total Preorder Relation.

common point: both are written using functors, showing how Cameleer proofs can scale up to the level of modular algorithms and data structures.

3.1 A Simple Functional Program the Merge Routine

Modular Definitions, Using Functors. The OCaml module system, in particular *functors*, offers flexible mechanisms to derive implementations that are agnostic to the actual representation of manipulated data types. Functors stand for modules that take other modules as parameters, similar to Scala traits. As an introductory example on the use of functors, consider the following implementation of a max function within functor Max:

```
module Max (E : PRE_ORD) = struct
  let max x y =
    if E.leq x y then y
    else x
end
```

The signature type PRE_ORD is given in Fig. 3. It introduces a type t together with a function leq, which establishes a total preorder on values of type t. The GOSPEL specification in this module type introduces a predicate le, which we assume it respects the three laws of a preorder: *reflexivity*, *totality*, and *transitivity*. Such laws are encoded as *axioms*, which stand for logical assumptions upon which one relies without actually providing them correct. Finally, the postcondition of program function leq states this is a decidable implementation of predicate le.

For the above implementation of max, we use the leq function provided in the functor argument E, to check whether function argument x is less or equal to y. This comparison is made with respect to the preorder relation induced by E. We can provide this implementation with suitable GOSPEL specification. This is as follows[2]:

```
let max x y = ...
(*@ r = max x y
      ensures E.le x r /\ E.le y r
      ensures r = x \/ r = y
```

The first clause in the postcondition states that both `x` and `y` must be smaller than or equal to the returned value `r`, with respect to the preorder induced by predicate `E.le`. The second clause states that `r` must be either equal `x` or `y`, where `(=)` stands for polymorphic, potentially undecidable, logical equality.

OCaml Implementation. The *merge sort* algorithm gets its name from its main step: merging the elements of two sorted lists, producing a third sorted list. Here, our goal is to provide a `merge` implementation *independently* of the type of list elements. To do so, we introduce the following functor `Merge`:

```
module Merge (E : PRE_ORD) = struct
  type elt = E.t

  let rec merge_aux acc l1 l2 =
    match (l1, l2) with
    | [], l | l, [] -> List.rev_append acc l
    | x :\,\!: xs, y :\,\!: ys ->
        if E.leq x y then merge_aux (x :\,\!: acc) xs l2
        else merge_aux (y :\,\!: acc) l1 ys

  let merge l1 l2 = merge_aux [] l1 l2
end
```

The above `merge` definition is an efficient implementation of a merge routine, since it makes use of the tail-recursive, auxiliary function `merge_aux`. This function merges lists `l1` and `l2` into the accumulator `acc`. Since every new element is inserted to the head of `acc`, then in the base cases one must first reverse it and then concatenate the result with `l`, the suffix list of elements (either from `l1` or `l2`) that remains to be enumerated. The OCaml standard library function `rev_append` efficiently implements this "reverse then concatenate" process. Finally, the main `merge` function calls `merge_aux` with the empty list as the initial value for the accumulator.

GOSPEL Specification. Let us now describe the specification of the `merge_aux` and `merge` functions. In order to specify that `merge` always returns a sorted list, we must first introduce what it means for a list to be sorted. We introduce the following GOSPEL predicate:

```
(*@ predicate rec sorted_list (l : elt list) =
      match l with
```

[2] In Cameleer, function specification is introduced after function definition.

```
      | [] | _ :\,\!: [] -> true
      | x :\,\!: y :\,\!: r -> E.le x y && sorted_list (y :\,\!:
   r) *)
(*@ variant l *)
```

The empty or singleton lists of integers are always sorted. If the list has at least two elements, then the first must less than or equal to the second one and the suffix list `y :\,\!: r` must also be a sorted list. Since `sorted_list` is to be used within specifications, it must be a total (*i.e.*, terminating) function. We supply the variant `l`, which represents a *termination measure* for every recursive call to the `sorted_list` predicate. A variant represents a quantity that always strictly decreases at every recursive call. In this case, we state that the argument of every recursive call is structurally smaller than the value of `l` at the entry point.

Note that the elements of the argument `l` are of type `integer`, the GOSPEL type for mathematical integers. When applying this function to a list of values of OCaml `int` type (63-bit machine integers), the GOSPEL and Cameleer tool-chains will apply a conversion mechanism from machine integers into their equivalent mathematical representation.

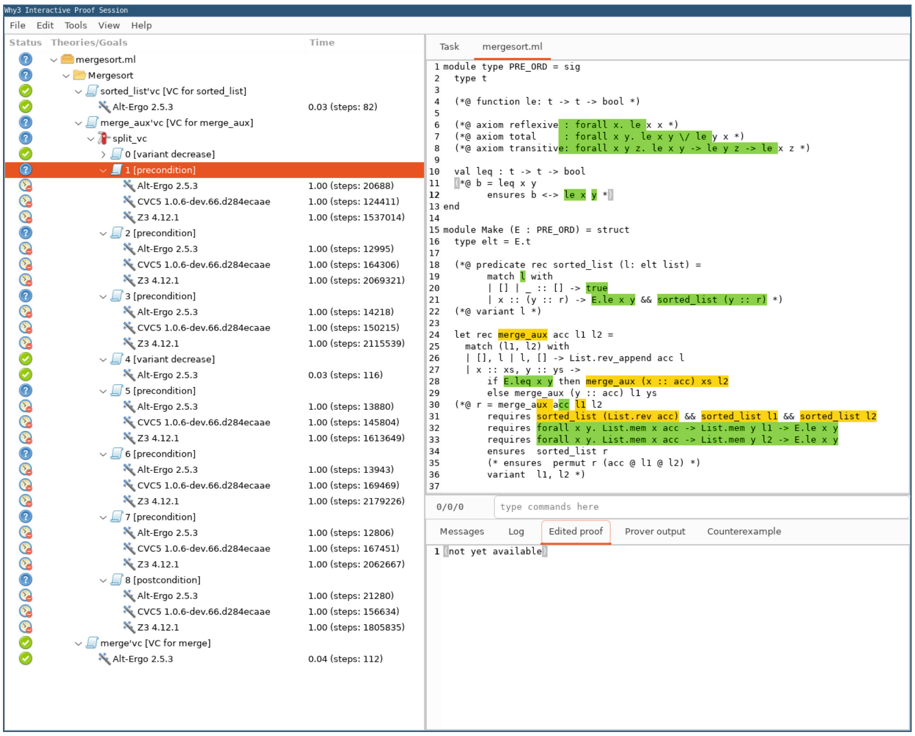
Using `sorted_list` predicate, we attach the following GOSPEL specification to the `merge_aux` function:

```
let rec merge_aux acc l1 l2 = ...
(*@ r = merge_aux acc l1 l2
      requires sorted_list (List.rev acc)
      requires sorted_list l1 && sorted_list l2
      requires forall x y.
        List.mem x acc -> List.mem y l1 -> E.le x y
      requires forall x y.
        List.mem x acc -> List.mem y l2 -> E.le x y
      ensures  sorted_list r
      variant  l1, l2 *)
```

The precondition reads as follows: the `acc` list is sorted in reverse order, while `l1` and `l2` are sorted in natural order; every element from `acc` must be less or equal to any element from either `l1` or `l2`. Finally, the postcondition simply asserts the returned list `r` is sorted and we prove termination using the lexicographic order on the pair `l1, l2`. In other words, if in a recursive call `l1` structurally decreases, then the whole variant decreases; otherwise, when `l1` does not decrease, then it must be the case that `l2` decreases. Cameleer ships with a subset of the OCaml standard library specified using GOSPEL, hence one is able to use and reason about functions such as `List.mem` or `List.rev`.

Finally, the specification of `merge` is as follows:

```
let merge l1 l2 = merge_aux [] l1 l2
(*@ r = merge l1 l2
      requires sorted_list l1 && sorted_list l2
      ensures  sorted_list r *)
```



```
Status  Theories/Goals                                  Time
?       mergesort.ml
?         Mergesort
✓           sorted_list'vc [VC for sorted_list]
✓             Alt-Ergo 2.5.3                            0.03 (steps: 82)
?           merge_aux'vc [VC for merge_aux]
?             split_vc
✓               0 [variant decrease]
?               1 [precondition]
                  Alt-Ergo 2.5.3                        1.00 (steps: 20688)
                  CVC5 1.0.6-dev.66.d284ecaae           1.00 (steps: 124411)
                  Z3 4.12.1                             1.00 (steps: 1537014)
?               2 [precondition]
                  Alt-Ergo 2.5.3                        1.00 (steps: 12995)
                  CVC5 1.0.6-dev.66.d284ecaae           1.00 (steps: 164306)
                  Z3 4.12.1                             1.00 (steps: 2069321)
?               3 [precondition]
                  Alt-Ergo 2.5.3                        1.00 (steps: 14218)
                  CVC5 1.0.6-dev.66.d284ecaae           1.00 (steps: 150215)
                  Z3 4.12.1                             1.00 (steps: 2115539)
✓               4 [variant decrease]
✓                 Alt-Ergo 2.5.3                        0.03 (steps: 116)
?               5 [precondition]
                  Alt-Ergo 2.5.3                        1.00 (steps: 13880)
                  CVC5 1.0.6-dev.66.d284ecaae           1.00 (steps: 145804)
                  Z3 4.12.1                             1.00 (steps: 1613649)
?               6 [precondition]
                  Alt-Ergo 2.5.3                        1.00 (steps: 13943)
                  CVC5 1.0.6-dev.66.d284ecaae           1.00 (steps: 169469)
                  Z3 4.12.1                             1.00 (steps: 2179226)
?               7 [precondition]
                  Alt-Ergo 2.5.3                        1.00 (steps: 12806)
                  CVC5 1.0.6-dev.66.d284ecaae           1.00 (steps: 167451)
                  Z3 4.12.1                             1.00 (steps: 2062667)
?               8 [postcondition]
                  Alt-Ergo 2.5.3                        1.00 (steps: 21280)
                  CVC5 1.0.6-dev.66.d284ecaae           1.00 (steps: 156634)
                  Z3 4.12.1                             1.00 (steps: 1805835)
✓           merge'vc [VC for merge]
✓             Alt-Ergo 2.5.3                            0.04 (steps: 112)
```

```
module type PRE_ORD = sig
  type t

  (*@ function le: t -> t -> bool *)

  (*@ axiom reflexive : forall x. le x x *)
  (*@ axiom total     : forall x y. le x y \/ le y x *)
  (*@ axiom transitive: forall x y z. le x y -> le y z -> le x z *)

  val leq : t -> t -> bool
  (*@ b = leq x y
        ensures b <-> le x y *)
end

module Make (E : PRE_ORD) = struct
  type elt = E.t

  (*@ predicate rec sorted_list (l: elt list) =
        match l with
        | [] | _ :: [] -> true
        | x :: (y :: r) -> E.le x y && sorted_list (y :: r) *)
  (*@ variant l *)

  let rec merge_aux acc l1 l2 =
    match (l1, l2) with
    | [], l | l, [] -> List.rev_append acc l
    | x :: xs, y :: ys ->
        if E.leq x y then merge_aux (x :: acc) xs l2
        else merge_aux (y :: acc) l1 ys
  (*@ r = merge_aux acc l1 l2
        requires sorted_list (List.rev acc) && sorted_list l1 && sorted_list l2
        requires forall x y. List.mem x acc -> List.mem y l1 -> E.le x y
        requires forall x y. List.mem x acc -> List.mem y l2 -> E.le x y
        ensures  sorted_list r
        (* ensures  permut r (acc @ l1 @ l2) *)
        variant  l1, l2 *)

```

```
0/0/0    type commands here
```

```
1 (not yet available)
```

Fig. 4. Why3 Proof Session for the Merge Sort Routine.

If both `l1` and `l2` are sorted lists, then the call `merge l1 l2` always produces a sorted list.

Cameleer Proof. Assuming the OCaml implementation and GOSPEL specification of the merge routine are contained in file `merge.ml`, one can start the verification process by typing the command `cameleer merge.ml`. This launches the interactive Why3 graphical user interface [10], as depicted in Fig. 4. On left-hand side, the Why3 IDE features a node for the generated verification conditions (VCs) of each top-level definition, together with any proof attempt on such VCs. Calling a SMT solver to prove a generated VC can be done by right-clicking a node and selecting the desired solver. A green button on the left of a node means that a solver was able to discharge the corresponding VC. This is the case of the `sorted_list` predicate and the `merge` function, for which Alt-Ergo is able to prove that both adhere to their specification, in less than a second. The proof time is shown on the right of the solver name, together with the number of conducted proof steps. One can also apply proof transformations on nodes, for instance to split a larger VC into its conjunctive clauses. This is done for the `merge_aux` definition. After split, one can focus on a specific part of the verification, such as proving a precondition, variant decrease, or postcondition.

Moreover, each of these individual formulae is smaller and less involved than the original VC, hence more likely to be automatically discharged by an SMT solver.

On the right-hand side of the Why3 IDE, one can inspect the code under verification. On one hand, the green labels stand for the assumptions (flow of the execution and parts of the specification) made at some point of the verification process. On the other hand, the yellow labels mark what one is actually trying to verify. In this case, we are trying to discharge the first precondition of the `merge_aux` function, at the recursive call `merge_aux (x :,!: acc) xs l2`. We shall explain the `Task` tab in Sect. 3.2.

As shown in Fig. 4, we fail to prove that function `merge_aux` adheres to its GOSPEL specification. We are not able to verify every individual VC for this function, after split. The three used solvers, Alt-Ergo, CVC5, and Z3, all time-out after 1 s for every condition except that the supplied variant measure decreases. One could attempt to provide more time to each solver, to allow these tools to conduct more proof steps, hopefully leading to more VCS being discharged. However, in this case, we are actually missing some *auxiliary lemmas* about the `sorted_list` predicate. A lemma in GOSPEL represents a property that, once stated, can be explored by SMTs to discharge other VCs. However, contrarily to axioms, lemmas are not assumed: these must be proved correct at definition time.

To close the verification of the `merge_aux` function, we need two auxiliary lemmas: first, one that states we can insert a new value `x` as the head of a sorted list `l` if and only if `x` is less than or equal to every element in `l`. We introduce such a lemma in GOSPEL as follows:

```
(*@ lemma sorted_mem: forall x l.
      (forall y. List.mem y l -> E.le x y) /\ sorted_list l <->
      sorted_list (x :\,\!: l) *)
```

Second, the concatenation `l1 @ l2` of sorted lists `l1` and `l2` is a sorted list if and only if all the elements in `l1` are less or equal than all the elements of `l2`. We provide the following GOSPEL lemma:

```
(*@ lemma sorted_append: forall l1 l2.
      (sorted_list l1 && sorted_list l2 &&
       (forall x y. List.mem x l1 -> List.mem y l2 -> E.le x y))
  <-> sorted_list (l1 ++ l2) *)
```

Using the given lemmas, the correctness proof for `merge_aux` now succeeds. Alt-Ergo is able to explore these auxiliary definitions to discharge all the remaining VCs. As for the proofs of the lemmas themselves, both require proofs by induction. We can conduct such proofs inside the Why3 IDE, using a dedicated transformation for induction over algebraic types (in this case lists). For a more complete presentation on how to use the Why3 IDE, including on how to apply different interactive proof transformations, we refer the reader to the framework user's manual [2].

```
module type EQUAL = sig
  type t

  val eq : t -> t -> bool
  (*@ b = eq x y
        ensures b <-> x = y *)
end
```

Fig. 5. Type Equipped With an Equality Relation.

3.2 Searching an Element Within an Array

OCaml Implementation. We now make a shift from the purely functional world to present some OCaml imperative features, and showcase how one can use Cameleer to reason about such features. Consider the following modular implementation of a function that performs a linear search in an array:

```
module Find (E : EQUAL) = struct
  let find x a =
    let exception Found of int in
    try
      for i = 0 to Array.length a - 1 do
        if E.eq a.(i) x then raise (Found i)
      done;
      raise Not_found
    with Found i -> i
end
```

Fig. 5 presents the definition of the signature type `EQUAL`. It declares a function `eq` that decides whether two values of type `t` are *logically equal*. This is exactly what is stated in the postcondition clause.

The definition of the `find` function presents some interesting OCaml imperative traits. On one hand, the use of a `for` loop to scan the array `a`; on the other hand, the declaration and use of the *local exception* `Found`. The latter is used to signal that the search succeeded, carrying the index of `x` within `a`. The use of local exceptions in OCaml is a convenient way to simulate the behavior of a `return` statement, commonly found in other languages. In fact, the whole loop is surrounded with a `try..with` block that ensures exception `Found` is always caught. Finally, to signal that `x` does not occur in `a`, we use the `Not_found` exception from the OCaml standard library. It is worth noting that we purposely let such an exception escape the scope of `find`.

Cameleer Proof. In order to prove the correctness of function `find`, one must supply a *loop invariant*. This is done in Cameleer as follows:

```
for i = 0 to Array.length a - 1 do
  (*@ invariant forall j. 0 <= j < i -> a.(j) <> x *)
```

This invariant simply asserts that while in the loop, we know for sure `x` does not occur in the prefix of `a` that we have already scanned. The infix operator `(<>)` stands for logical inequality. As for the equality operator, `(=)`, inequality is expressed using the same syntax in OCaml and in GOSPEL and both are built-in symbols of the GOSPEL language. We recall that, except for quantifiers and logical connectives, GOSPEL terms are a written in a subset of the OCaml language.

Now, we focus on providing a specification contract for function `find`. But first, it is crucial to distinguish the possible outcome behaviors of this function. On one hand, it returns normally whenever exception `Found` is raised; on the other hand, it raises the `Not_found` exception to abort execution. For the former, we shall establish a *regular postcondition*. For the latter, we shall introduce what is called an *exceptional postcondition*. We attach the following GOSPEL annotations to `find`:

```
let find x a =
  ...
(*@ i = find x a
      ensures a.(i) = x
      raises Not_found -> forall i. 0 <= i < Array.length a ->
        a.(i) <> x *)
```

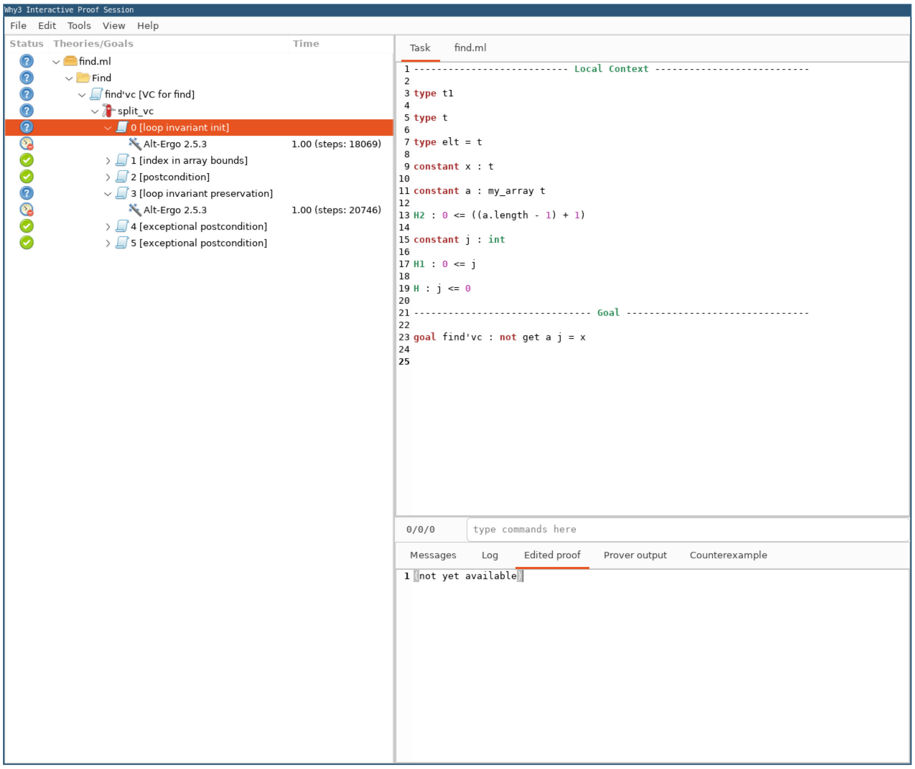
The `ensures` clause is checked when `find` indeed returns an integer `i`, representing the (first) index of `x` in `a`. The `raises` clause states the logical property that holds when `Not_found` is raised. This stands for the case when we have scanned all the array `a`, finding no occurrence of `x`. We restrict the range of values that the universally quantified variable `i` can take, since GOSPEL establishes that undefined array indices are arbitrary values, not necessarily different from `x`.

For this program, Cameleer generates 6 VCs, after splitting the formula generated for the `find` function. All are immediately discharged by Alt-Ergo.

Providing an Incorrect Loop Invariant. Let us take a step back in the verification process of the `find` implementation. Imagine a scenario where one would have, incorrectly, supplied the following loop invariant:

```
for i = 0 to Array.length a - 1 do
  (*@ invariant forall j. 0 <= j <= i -> a.(j) <> x *)
```

The only difference, when compared with the previously presented invariant, is that now the value of the universally quantified variable `j` can be equal to `i`, the loop index. Figure 6 shows that by feeding the new invariant to the Cameleer-Why3 pipeline, one is still able to prove the postcondition of `find` holds, but not the *invariant initialization* (*i.e.*, the invariant holds before the first iteration), neither *invariant preservation* (*i.e.*, assuming the invariant holds before an arbitrary iteration, it still holds after that iterations completes). To debug a failed proof attempt, the Why3 IDE allows the user to inspect the *task* [3], a representation of the formula that is sent to solvers. Under tab `Task`, such


Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Time
Task find.ml
0/0/0
type commands here
Messages Log Edited proof Prover output Counterexample

```
find.ml
  Find
    find'vc [VC for find]
      split_vc
        0 [loop invariant init]
          Alt-Ergo 2.5.3                       1.00 (steps: 18069)
        1 [index in array bounds]
        2 [postcondition]
        3 [loop invariant preservation]
          Alt-Ergo 2.5.3                       1.00 (steps: 20746)
        4 [exceptional postcondition]
        5 [exceptional postcondition]
```

```
--------------------------- Local Context ---------------------------

type t1

type t

type elt = t

constant x : t

constant a : my_array t

H2 : 0 <= ((a.length - 1) + 1)

constant j : int

H1 : 0 <= j

H : j <= 0

------------------------------ Goal --------------------------------

goal find'vc : not get a j = x


```

```
[not yet available]
```

Fig. 6. Proof Task for an Incorrect Loop Invariant Initialization.

formula is displayed as a `goal` (*i.e.*, what one is actually trying to prove) and the proof context (*i.e.*, the hypotheses) above the dashed line.

Figure 6 depicts the task for the loop invariant initialization. After reading the task, we can conclude that hypotheses `H1` and `H2` imply that `j` is equal to 0. Hence, we are trying to prove a goal that asserts the actual first element in the array `a` is not `x`. There is nothing in our proof context that allows us to prove such a statement. This is an indication that either our context is not enough to discharge the goal (*e.g.*, the specification is incomplete), or rather there is an actual bug in the specification or implementation. In this case, however, we already know the answer: changing the loop invariant to `j < i` would generate a task, for loop invariant initialization, with hypotheses `0 <= j` and `j < 0`, hence the goal would hold vacuously. It is worth noting that tasks are written in the logical fragment of the WhyML language. This is the only point in the verification process that a Cameleer user must read a formula that is not written in GOSPEL. However, since GOSPEL and WhyML are syntactically very similar, we believe someone familiar with GOSPEL is able to read and understand a Why3 task.

4 Purely Functional Programming

We define *purely functional programming* as the approach to write a program where no mutable state is involved. Such a program is normally a collection of (recursive) functions that are composed with each other to perform some computation. The program also only employs data structures that do not require any memory manipulation, such as lists or trees. This style of programming is at the very essence of the OCaml language.

In this section, we use the Cameleer tool to conduct formal verification of a purely functional program that operates on trees. We present the OCaml implementation of such a program, annotated with suitable GOSPEL specifications. We interleave the presentation of code listings with explanations of the given implementation and specification. Hence, to ease readability, the example chunks that belong together are given running line numbers.

4.1 Same Fringe Comparing Two Binary Trees

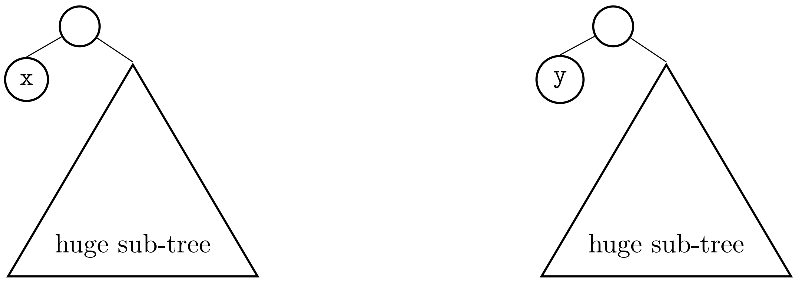
Let us consider the following, very classic, programming challenge:

> *Write a function that, given two binary trees, decides whether the two trees present the same sequence of elements when traversed inorder.*

This is known as the *same fringe* problem. One possible solution to this problem is as follows:

1. perform an inorder traversal on both trees, building the list of elements enumerated during such traversals;
2. compare, recursively, whether the two lists contain the same elements.

This is, however, a very naive approach, since it always builds the auxiliary lists for all the elements of both trees. Imagine the following scenario:



The two trees differ in the leftmost element, as we have x for the left-hand side tree, and y for the right-hand side tree. Following the solution proposed above, we would unnecessarily build two (huge) sequences. Note, however, that this implementation is easier to check for correctness than more efficient ones. Hence, this list-based approach can be used as a good specification for the solution we describe in the remaining of the section.

We propose to explore an approach that allows one to enumerate the elements of each tree, step-by-step. In such a way, we can stop as soon as two distinct elements are enumerated. If we complete the iteration process on both trees, then it must be the case the trees contain the same elements. This algorithm is implemented in OCaml and specified using GOSPEL as follows. First, we use the `EQUAL` signature from Fig. 5 to build a functor that implements *same fringe*. We start by defining the type of binary trees with elements of type `E.t`, as follows:

```
module Make (E : EQUAL) = struct
  type tree = Empty | Node of tree * E.t * tree
```

A `tree` is either `Empty` or a `Node` formed of two sub-trees and a root of type `E.t`.

Now, we define a logical function that implements an inorder traversal on a binary tree, returning the list of enumerated elements:

```
  (*@ function elements (t : tree) : E.t list =
        match t with
        | Empty -> []
        | Node (l, x, r) -> (elements l) @ (x :\,\!:
  elements r) *)
```

The traversal is implemented by: (i) traversing the whole left sub-tree; (ii) concatenating the resulting sequence of elements (the `(@)` operator represents list concatenation in OCaml) to `x` (the root) and the sequence of elements issued from the right sub-tree traversal. This is exactly the naive approach previously described. We shall only use the function `elements` for specification purposes.

In order to implement the step-by-step enumeration of elements in a tree, we use an explicit data representation of the call stack of the program that would construct the inoder list, so we can interrupt the traversal as soon as required. Such a representation is inspired by the *zipper* [16] structure.A zipper can be used as an efficient cursor into data structures, allowing one to arbitrarily traverse the structure, as well as to perform efficient local modifications (*e.g.*, insertions or deletions) without the overhead of rebuilding the whole structure for each modification. In the case of the *same fringe* problem, we always traverse a tree towards its leftmost element, without performing any modifications to the structure. Hence, we specialize the zipper data type as follows:

```
  type zipper = (E.t * tree) list
```

A value of type `zipper` is a list, where each element is a pair composed of a tree element and the corresponding right sub-tree. We build such a list bottom-up, which represents the left spine of the tree that is still to be traversed. Together with the `zipper` data type, we introduce the following logical function to convert from a `zipper` to a list:

```
  (*@ function enum_elements (e : zipper) : E.t list =
        match e with
        | [] -> []
        | (x, r) :\,\!: e -> x :\,\!: (elements r @
  enum_elements e) *)
```

From a specification point of view, we have everything we need to tackle our verified implementation of the *same fringe* problem. We start by defining how to create a zipper from a tree. This is done as follows:

```
let rec mk_zipper (t : tree) (e : zipper) =
  match t with
  | Empty -> e
  | Node (l, x, r) -> mk_zipper l ((x, r) :\,\!: e)
(*@ r = mk_zipper t e
      variant t
      ensures enum_elements r = elements t @ enum_elements
e *)
```

The specification of `mk_zipper` states that this is a terminating function, with argument `t` structurally decreasing at each recursive call. The functional behavior of this function is captured in the postcondition, where we state the sequence of elements of the resulting zipper is the same as the inorder sequence of elements from tree `t`, plus the elements of the accumulator `e`.

We now provide the actual, step-by-step, iteration on two zippers, as follows:

```
let rec eq_zipper (e1 : zipper) (e2 : zipper) =
  match (e1, e2) with
  | [], [] -> true
  | (x1, r1) :\,\!: e1, (x2, r2) :\,\!: e2 -> E.eq x1 x2
&&
      eq_zipper (mk_zipper r1 e1) (mk_zipper r2 e2)
  | _ -> false
 (*@ b = eq_num e1 e2
      variant List.length (enum_elements e1)
      ensures b <-> enum_elements e1 = enum_elements e2 *)
```

This implementation distinguishes three cases:

1. If both zippers are empty, then we are sure to have enumerated the same sequence of elements.
2. If both zippers still have elements, then the next elements in the enumeration are the heads of both lists. We compare these and proceed recursively, only if the `E.eq x1 x2` comparison holds.
3. Otherwise, if one of the zipper terminates before the other, then we are sure the enumerated sequences differ.

Very simply put, the postcondition of `eq_zipper` states that this function logically decides whether two zippers enumerate the same sequence of elements.

Finally, we provide the definition of the `same_fringe` function, which decides whether two binary trees present the same elements. This is as simple as

```
let same_fringe (t1 : tree) (t2 : tree) =
  eq_zipper (mk_zipper t1 []) (mk_zipper t2 [])
 (*@ b = same_fringe t1 t2
              ensures b <-> elements t1 = elements t2 *)

  end
```

From the postcondition of `eq_zipper` one can deduce that `same_fringe` will return the Boolean `true` if and only if the two zippers passed as arguments represent the same sequence of elements. Since we use the empty list as the initial accumulator value for the creation of both zippers, then the postcondition of `mk_zipper` states the created zippers enumerate the exact same sequence of elements as those of trees `t1` and `t2`, respectively. Hence, the postcondition of `eq_zipper` always holds.

Feeding our *same fringe* implementation to Cameleer generates 11 VCs, after splitting each top-level definition. These are discharged in roughly 1 s using a combination of the Alt-Ergo, Z3, and CVC5 SMT solvers. For reference, the complete OCaml implementation and GOSPEL specification for *same fringe* is given in the appendix of the extended version [29].

4.2 Summary

In this section, we used the *same fringe* example to motivate and showcase the use of Cameleer for the deductive verification of purely functional algorithms. Such functional implementations are closer to mathematical definitions, hence are normally easier to reason about. Other than the example presented in this section, Cameleer has been successfully used to prove the correctness of real-world functional OCaml data structures. We highlight the `Set` module from the OCaml standard library, and the Leftist Heap implementation issued from the widely used `ocaml-containers` library[3]. Even if not presented in the body of this document, all such case studies are included in the companion artifact.

5 Imperative Programs

One important aspect of the OCaml language is the fact that it is a multi-paradigm language, combining functional with imperative and object-oriented programming. In this section, we use Cameleer to conduct the formal verification of an OCaml program that implements an historical algorithm using imperative features, namely loops, mutable references, and exceptions. As in Sect. 4, the main example code listings are presented using running line numbers.

5.1 Boyer-Moore MJRTY Algorithm

Let us, once again, use an algorithmic problem as the vehicle to showcase how to specify an OCaml program using GOSPEL, and how to prove it in Cameleer. Consider the following challenge:

[3] https://github.com/c-cube/ocaml-containers.

Write a function that, given an array of votes, determines the candidate with the absolute majority, if any.

The more direct solution would take the following steps:

1. For a total of N candidates, we first allocate an array of integer values of length N that serves as an histogram.
2. We do a first pass over the array of votes, summing up in the histogram the number of votes for each candidate.
3. Finally, we iterate over the histogram to check if any of the candidates achieves majority.

This approach could certainly be implemented in OCaml and proved correct in Cameleer. It runs in $\mathcal{O}(M + N)$ time (build the histogram, then iterate over it), where M is the number of votes . However, this approach allocates an extra memory space of N cells, *i.e.*, the histogram. Here, we adopt a different solution, due to R. Boyer and J. Moore [4]. Such a solution uses at most $2M$ comparisons and constant extra space (other than the array of votes itself).

First, we build a functor parameterized with module type `EQUAL` from Fig. 5, so that we are able to compare candidates:

```
1   module Mjrty (E : EQUAL) = struct
2     type candidate = E.t
3
4     let mjrty a =
5       let exception Found of candidate in
```

We begin by declaring local exception `Found`, which we use to terminate the search and signal if some candidates reaches majority. We now introduce the only extra auxiliary references we need:

```
6       let n = Array.length a in
7       let cand = ref a.(0) in
8       let k = ref 0 in
```

The use of references `cand` and `k` is the actual core of the Boyer-Moore's method.

We do a first traversal on the array of votes, updating `cand` and `k` accordingly:

```
9       try
10        for i = 0 to n - 1 do
11          if !k = 0 then begin
12            cand := a.(i);
13            k := 1 end
14          else if E.eq !cand a.(i) then incr k
15          else decr k
16        done;
```

Very briefly, the loop body does the following:

– If the value stored in `k` is zero, then we change the candidate stored in `cand` to the i-th element of `a` and update `k` to one (lines 11 to 13);

– If the i-th candidate in `a` is equal to the value stored in `cand`, then we increment reference `k` (line 14);
– Otherwise, we decrement `k` (line 15).

So now, a crucial question arises: what are the invariants for this loop that allow verification to succeed? One crucial part of the process is to reason about "*the number of votes for a certain candidate in a slice of the array*". To be able to express such notion in the specification, we declare the following GOSPEL function:

```
(*@ function numof_eq (a : 'a array) (v : 'a) (l u: integer) :
      integer *)
```

This function represents the number of elements from `a`, within range [`l`; `u`), that are equal to value `v`. For now, we focus on using `numof_eq` to establish the loop invariants. Later in this section, we provide a proper definition for this function.

Let us a conduct a step-by-step analysis on how references `cand` and `k` are used together within the loop, as to derive the loop invariants. In fact, the invariants we present here are already given in Boyer and Moore's original work [4], and we adapt those into GOSPEL:

1. The number of votes for candidate `cand`, within the array prefix already scanned, is at least `k`. We write this down in GOSPEL as follows:

   ```
   (*@ invariant 0 <= !k <= numof_eq a !cand 0 i
   ```

   This invariant is maintained by the case analysis implemented from line 11 to 15 in the code snippet above: we update the value of `cand` whenever `!k` reaches zero, hence `!k` never stores a negative number; we update the value stored in `k`, without changing the candidate, hence `k` is kept as a lower bound for the actual number of occurrences of `!cand` in the scanned part of the array. In other words, we are sure we only decrement `k` after a sufficient number of increments occurred (except if we decrement after reference `cand` is updated, but in this case `k` is update to one, hence an implicit increment has also occurred).
2. The actual number of votes for `cand` minus the value of `k` cannot exceed `(i - !k)/2`:

   ```
       invariant 2 * (numof_eq a !cand 0 i - !k) <= i - !k
   ```

   Instead of a division, we write such an invariant using an equivalent multiplication. The reason is somehow low-level and tied to the upcoming verification effort: SMT solvers are known to handle multiplication better than division.
3. For every candidate `c` other than `cand`, the number of votes for `c`, within the scanned prefix of the array, cannot exceed `(i - !k)/2`:

   ```
       invariant forall c. c <> !cand ->
                    2 * numof_eq a c 0 i <= i - !k *)
   ```

   This invariant implies that no other candidate, other than the one stored in `cand`, can have the majority of votes in the slice of the array already processed by the algorithm. Once again, we use a multiplication by two to avoid the division.

The last invariant actually allows one to deduce a crucial property: after scanning all the array, cand is the only candidate that can effectively reach majority.

After the loop, we immediately check whether we are in position to provide a final answer:

```
        if !k = 0 then raise Not_found;
        if 2 * !k > n then raise (Found !cand);
```

If k stores zero, we are sure no candidate has reached majority. We use the OCaml standard library Not_found exception to signal such behavior. If the value stored in k is more than half of n, the size of the array, we are sure cand has reached majority. We use locally-defined exception Found to signal this behavior. If none of the above conditions is met, then we cannot give yet a definitive answer; we need an extra traversal over the array to check whether cand has the majority.

The final step of the implementation is a simple loop that counts the actual number of votes for cand. If, at some point in the traversal, the accumulated votes for cand are greater then half of n, we terminate signaling the majority of this candidate. We can then reuse reference k for the purpose of counting votes:

```
        k := 0;
        for i = 0 to n - 1 do
         (*@ invariant !k = numof_eq a !cand 0 i && 2 * !k <=
    n *)
          if E.eq a.(i) !cand then begin
            incr k;
            if 2 * !k > n then raise (Found !cand) end
        done;
        raise Not_found
      with Found c -> c
```

This loop invariant states that reference k stores the actual number of occurrences of cand and that, if we keep iterating, then it must be the case that the k does not yet represent the majority of votes. If we reach past the loop, then cand does not have the majority, neither does any other candidate. Once again, we use Not_found to signal such an outcome.

The final piece in our verified OCaml implementation of the Boyer-Moore algorithm is the actual specification for function mjrty. This is as follows:

```
    (*@ c = mjrty a
          requires 1 <= Array.length a
          ensures  2 * numof_eq a c 0 (Array.length a) >
                   Array.length a
          raises   Not_found -> forall x.
                     2 * numof_eq a x 0 (Array.length a) <=
                          Array.length a *)
   end
```

As a precondition, we assume that the input array has at least one element. For the regular postcondition, *i.e.*, the one reached by catching exception `Found`, we prove that the returned candidate indeed has the absolute majority of votes. Finally, in the exceptional postcondition (*i.e.*, the one reached by raising exception `Not_found`), we prove that no candidate has enough votes to reach majority.

Definition of `numof_eq` Function. To conclude our proof of `mjrty` implementation, we must provide an actual definition for function `numof_eq`. We do so by means of an auxiliary function `numof`. This is defined, in GOSPEL, as follows:

```
(*@ function rec numof (p : integer -> bool) (a b : integer) :
      integer
    = if b <= a then 0 else
      if p (b - 1) then 1 + numof p a (b - 1)
                   else     numof p a (b - 1) *)
(*@ variant b - a *)
```

The call `numof p a b` returns the number of integer values, within a certain range $[\texttt{a}; \texttt{b})$, that satisfy a given predicate `p`. We attach the variant `b - a` to the above definition, which allows us to prove this is a total function.

To define `numof_eq`, we specialize `numof` for arrays and an equality predicate:

```
(*@ function numof_eq (a : 'a array) (v : 'a) (l u : integer) :
      integer
    = numof (fun j -> a.(j) = v) l u *)
```

The use of the higher-order function `numof` leads to a concise and elegant definition for `numof_eq`. This is in an interesting application of functional programming concepts to derive sound, expressive, and yet intuitive specifications even in the presence of mutable data structures.

The right-to-left definition of `numof` is useful when it comes to proving the preservation of loop invariants where one is scanning an array from left to right. At the beginning of the i-th iteration, one assumes that `numof p 0 i` represents the number of elements, in the slice $[0; \texttt{i})$ of some array, that respect predicate `p`. At the end of the iteration, we must re-establish the invariant for the range $[0; \texttt{i} + 1)$, *i.e.*, `numof p 0 (i + 1)`. If the i-th element respects predicate `p`, then we take the `then` branch in the definition of `numof`; otherwise, we take the `else` branch. In either cases, the invariant is re-established simply following the definition of `numof`, since the recursive call `numof p 0 i` is exactly what we assumed at the beginning of the iteration. This applies to the proof of invariant preservation for the loops in the `mjrty` function, where `numof_eq a !cand 0 i` is mapped into a call to `numof (fun j -> a.(j) = !cand) 0 i`.

Finally, we provide auxiliary lemmas about the behavior of the `numof` function that allows us to close the proof of the `MJRTY` algorithm. First, we establish the lower and upper bound for the result of `numof`. A call `numof a b p`, for any given integer values `a` and `b` and a predicate `p`, always returns a non-negative value and cannot exceed `b - a`. This is captured by the following lemma:

```
(*@ lemma numof_bounds :
      forall p : (integer -> bool), a b : integer.
      a < b -> 0 <= numof p a b <= b - a *)
```

This lemma is proved interactively by induction on `b`, starting from `a`.

The next lemma states that a call to `numof p a c` can be written as the sum of calling `numof` in the range $[\mathtt{a};\mathtt{b}[$ and calling `numof` in the range $[\mathtt{b};\mathtt{c}[$, provided that $\mathtt{a} \leq \mathtt{b} \leq \mathtt{c}$. This is expressed as follows:

```
(*@ lemma numof_append:
    forall p: (integer -> bool), a b c: integer.
    a <= b <= c -> numof p a c = numof p a b + numof p b c *)
```

This lemma is proved by induction on `c`, starting from `a`.

The last two lemmas capture what happens in a single computation step of a call `numof p l u`, when `l < u`. On one hand, if value `l` respects predicate `p`, then we add 1 to the result of the recursive call `numof p (l + 1) b`:

```
(*@ lemma numof_left_add :
      forall p : (integer -> bool), l u : integer.
      l < u -> p l -> numof p l u = 1 + numof p (l + 1) u *)
```

On the other hand, if `l` does not respect `p`, then `numof p a b` is simply the result of the recursive call:

```
(*@ lemma numof_left_no_add:
      forall p : (integer -> bool), l u : integer.
      l < u -> not p l -> numof p l u = numof p (l + 1) u *)
```

One can also think of these lemmas as establishing the equivalence between either counting the number of elements that satisfy a given predicate from left-to-right, or from right-to-left. Both lemmas are proved by instantiating lemma `numof_append`, where `a` is instantiated with `l`, `b` with `l + 1`, and `c` with `u`.

The Cameleer-Why3 pipeline generates a total of 25 VCs for function `mjrty`. These are discharged using a combination of the Alt-Ergo, Z3, and CVC5 solvers. The proof of the auxiliary lemmas is also carried in the Why3 IDE, using dedicated transformations for induction over integer numbers and instantiating other lemmas. For reference, the complete OCaml implementation, GOSPEL specification, and auxiliary definitions for the `MJRTY` algorithm are given in appendix [29].

5.2 Summary

In this section, we showed how to use the imperative traits of OCaml to write elegant and efficient code. Moreover, with Cameleer, we are still able to prove the correctness of such implementations. The gallery of Cameleer verified programs includes several examples of verified imperative implementations. From those, we highlight the verification of a `Union Find` data structure, encoded in an array of integer values. An important feature of this case study is the use of the *decentralized invariants* technique [12] to achieve a fully-automated proof.

6 Related Work

Deductive software verification is now a mature discipline that is taught, worldwide, in the vast majority of Computer Science curricula. However, a large corpus of pedagogical, practical, hands-on oriented bibliography is still missing. One can cite the recent book on Dafny [20] as valuable contribution to fill this gap. On the other end of the spectrum of verification tools, the book by Nipkow *et al.* [26] and the Software Foundations volumes 3 [1] and 6 [9] provide comprehensive collections of data structures and algorithms formally verified in proof assistants. The first is completely developed in Isabelle, the other two in Coq.

When it comes to deductive verification of programs written in functional languages, one can cite frameworks like Iris [18] and Hoare Type Theory [25]. These are built on Coq, on top of very rich reasoning logics based on Separation Logic. These can scale up to the verification of complex imperative and concurrent programs. However, proofs in such frameworks are conducted manually, requiring a high degree of human interaction and proof expertise. In the particular case of verification of OCaml programs, the CFML [5] tool takes as input an OCaml program and translates it into a Coq term that captures the semantics of the program. The proof is then conducted using Separation Logic. CFML proofs are laborious and, in particular when compared with Cameleer, require extensive human interaction.

7 Conclusions and Future Perspectives

In this tutorial paper, we presented the deductive verification of different OCaml programs, ranging from purely functional implementations to code combining imperative features, such as mutable state and local exceptions. We use the Cameleer tool to conduct our practical experiments. This tool takes as input an actual OCaml implementation and translates it into an equivalent WhyML program, the language of the Why3 verification framework. Cameleer avoids the need to re-write entire OCaml code bases, just for sake of verification, as it would be the case with a direct use of Why3: one would have first to write a WhyML implementation and specification, then rely on an extraction mechanism to get an executable equivalent OCaml program. On the other hand, Cameleer is conceived with a clear focus towards proof automation, improving on the experience of entirely conducting manual proofs in an interactive proof assistant.

Throughout the paper, we use GOSPEL to attach formal specification to OCaml programs. Our experience suggests that this language is a good compromise when it comes to conciseness and readability of specifications, without sacrificing rigor. This is a major argument to bring even more OCaml programmers to adopt formal methods techniques in their daily routines. Finally, GOSPEL can be used not only for deductive verification but also for dynamically analyze OCaml code. As future work, it would be interesting to collaboratively use static and dynamic analysis techniques to tackle the verification of different parts of a big piece of OCaml software, resorting to GOSPEL as the aggregation entity.

Acknowledgments. I sincerely thank the anonymous reviewers from the Formal Methods 2024 Tutorial Track. Their comments and suggestions have greatly improved the presentation of this paper.

Data Availability Statement. The artifact supporting the experiments of this paper is publicly available at Zenodo, https://doi.org/10.5281/zenodo.12588707.

References

1. Appel, A.W.: Verified Functional Algorithms, Version 1.5.4., vol. 3. Software Foundations (2023). http://softwarefoundations.cis.upenn.edu
2. Bobot, F., Filliâtre, J.C., Marché, C., Melquiond, G., Paskevich, A.: The Why3 Platform, Version 1.7. University Paris-Saclay, CNRS, Inria (2024). https://www.why3.org/doc/
3. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, pp. 53–64 (2011)
4. Boyer, R.S., Moore, J.S.: MJRTY: a fast majority vote algorithm. In: Boyer, R.S. (ed.) Automated Reasoning: Essays in Honor of Woody Bledsoe, pp. 105–118. Kluwer Academic Publishers, Dordrecht, Netherlands (1991)
5. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 418–430 (2011). https://doi.org/10.1145/2034773.2034828
6. Charguéraud, A.: Separation Logic for Sequential Programs (Functional Pearl). Proc. ACM Program. Lang. **4**(ICFP) (2020).https://doi.org/10.1145/3408998
7. Charguéraud, A.: A modern eye on separation logic for sequential programs. (Un nouveau regard sur la Logique de Séparation pour les programmes séquentiels) (2023). https://tel.archives-ouvertes.fr/tel-04076725
8. Charguéraud, A., Filliâtre, J.C., Lourenço, C., Pereira, M.: GOSPEL—providing OCaml with a formal specification language. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 484–501. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_29
9. Charguéraud, A.: Separation Logic Foundations, Version 2.0, vol. 6. Software Foundations (2024). http://softwarefoundations.cis.upenn.edu

10. Dailler, S., Marché, C., Moy, Y.: Lightweight interactive proving inside an automatic program verifier. In: 4th Workshop on Formal Integrated Development Environment (F-IDE) (2018)
11. Filliâtre, J.C.: Deductive software verification. Int. J. Softw. Tools Technol. Transf. **13**(5), 397–403 (2011). https://doi.org/10.1007/s10009-011-0211-0
12. Filliâtre, J.: Simpler proofs with decentralized invariants. J. Log. Algebraic Methods Program. **121**, 100645 (2021). https://doi.org/10.1016/J.JLAMP.2021.100645
13. Filliâtre, J.-C., Pascutto, C.: Ortac: runtime assertion checking for OCaml (Tool Paper). In: Feng, L., Fisman, D. (eds.) RV 2021. LNCS, vol. 12974, pp. 244–253. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88494-9_13
14. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
15. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.J.: Behavioral interface specification languages. ACM Comput. Surv. **44**(3), 16:1–16:58 (2012).https://doi.org/10.1145/2187671.2187678
16. Huet, G.P.: The zipper. J. Funct. Program. **7**(5), 549–554 (1997). https://doi.org/10.1017/S0956796897002864
17. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
18. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: a modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151
19. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for java. ACM SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (2006). https://doi.org/10.1145/1127878.1127884
20. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
21. Maksimović, P., Ayoun, S.É., Santos, J.F., Gardner, P.: Gillian, Part II: real-world verification for JavaScript and C. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 827–850. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_38
22. Meyer, B.: Eiffel: The Language. Prentice-Hall (1991). http://www.eiffel.com/doc/#etl
23. Monin, J.: Understanding Formal Methods. Springer, Verlag, London (2003). https://doi.org/10.1007/978-1-4471-0043-0_8
24. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
25. Nanevski, A., Morrisett, J.G., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Funct. Program. **18**(5–6), 865–911 (2008). https://doi.org/10.1017/S0956796808006953
26. Nipkow, T., et al.: Functional algorithms, verified (2021)
27. Pereira, M., Ravara, A.: Cameleer: a deductive verification tool for OCaml. arXiv preprint arXiv:2104.11050 (2021)

28. Pereira, M.: Practical Deductive Verification of OCaml Programs (2024). https://doi.org/10.5281/zenodo.12588707
29. Pereira, M.: Practical deductive verification of OCaml programs (extended version). arXiv preprint arXiv:2404.17901 (2024)
30. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74. LICS '02, IEEE Computer Society, USA (2002)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Software Verification with CPAchecker 3.0: Tutorial and User Guide

Daniel Baier, Dirk Beyer, Po-Chun Chien, Marie-Christine Jakobs, Marek Jankola, Matthias Kettl, Nian-Ze Lee, Thomas Lemberger, Marian Lingsch-Rosenfeld, Henrik Wachowitz, and Philipp Wendler

LMU Munich, Munich, Germany

https://cpachecker.sosy-lab.org

Abstract. This tutorial provides an introduction to CPAchecker for users. CPAchecker is a flexible and configurable framework for software verification and testing. The framework provides many abstract domains, such as BDDs, explicit values, intervals, memory graphs, and predicates, and many program-analysis and model-checking algorithms, such as abstract interpretation, bounded model checking, Impact, interpolation-based model checking, k-induction, PDR, predicate abstraction, and symbolic execution. This tutorial presents basic use cases for CPAchecker in formal software verification, focusing on its main verification techniques with their strengths and weaknesses. An extended version also shows further use cases of CPAchecker for test-case generation and witness-based result validation. The envisioned readers are assumed to possess a background in automatic formal verification and program analysis, but prior knowledge of CPAchecker is not required. This tutorial and user guide is based on CPAchecker in version 3.0. This user guide's latest version and other documentation are available at https://cpachecker.sosy-lab.org/doc.php.

Keywords: CPAchecker · Configurable Program Analysis · Formal Verification · Model Checking · Software Verification · Program Analysis · Testing · Tutorial · Correctness Certification · Witnesses · Witness Validation · Fault Visualization

1 Introduction

CPAchecker [35] is a framework for configurable software verification with a focus on the verification of C programs. It is based on the concept of configurable program analysis [26, 28, 29] and provides an extensive collection of verification algorithms and abstract domains. Throughout the past years, CPAchecker has been a top contender in the International Competition on Software Verification [11, 12, 13] and has helped identify over 240 bugs in Linux device drivers [45, 64, 84].

An extended version of this user guide is available in a technical report [7].

© The Author(s) 2025

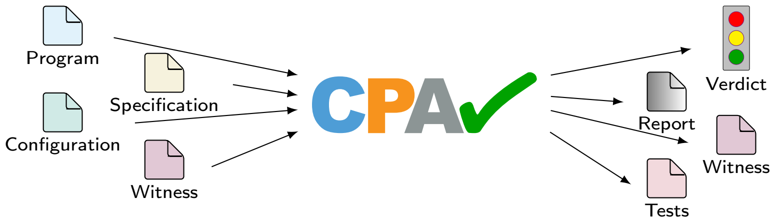


Fig. 1: Inputs and outputs of CPAchecker when it is used as a verifier, witness validator, or test-case generator

CPAchecker is open source and written in Java. Founded in 2007 at Simon Fraser University, it is now maintained by an active community (project statistics can be found on OpenHub.net). It puts a high priority on extensibility and flexible reuse of components for developers. The architecture and features of the framework are described in other articles [35, 48]. More information about the achievements, history, and license of CPAchecker are available in the extended version [7].

1.1 Use Cases of CPAchecker

There are three main use cases of CPAchecker, with their inputs and outputs summarized in Fig. 1: (1) As a *verifier*, CPAchecker takes as input a program and a specification, and returns a verdict, a verification report, and a verification witness. The verdict specifies whether the given program adheres to the specification, the verification report allows users to examine the verification result, and the witness contains a machine-readable justification for the returned verdict. (2) As a *witness validator* [5, 19], CPAchecker takes as input a program, a specification, and a witness, and returns a verdict that indicates whether the witness could be confirmed by CPAchecker. (3) As a *test-case generator* [32, 52, 75], CPAchecker takes as input a program and a test-coverage specification, and returns a set of test cases that cover the program according to the specification.

CPAchecker is also used for program transformation [31, 33, 34, 41, 42], to explore decompositions of verification problems [4, 27, 37], and to parallelize verification approaches [23, 37]. This tutorial focuses on using CPAchecker as a verifier. Information about CPAchecker as a witness validator and test-case generator is present in the extended version [7].

1.2 Configurable Program Analysis

CPAchecker uses configurable program analysis (CPA) [26, 28, 29] to compute a program's reachable states. A CPA specifies an abstract domain and a precision used to explore a program's reachable states. The abstract domain defines the representation of a program's state, while the precision defines how precise the abstraction should be. Various CPAs have been implemented in CPAchecker, each tailored to handle specific program features and perform a dedicated analysis. CPAs can also be combined to achieve synergy. Furthermore, precisions can be adjusted dynamically [29], making an analysis coarse but efficient, or precise but resource-consuming. CPAchecker automatically adjusts the precisions via counterexample-guided abstraction refinement (CEGAR) [24, 43, 44, 53] or some carefully-designed procedures [15].

1.3 Documentation and Communication

The README and directory doc/ in the CPAchecker project provide useful information for users and developers. For an overview on the architecture, we recommend the tool paper [35] on CPAchecker and the publications regarding the CPA concept [26, 28, 29]. CPAchecker supports various verification algorithms and techniques. The most important techniques in CPAchecker are explained in separate publications, including data-flow and value analysis [15, 26, 43], SMT-based verification algorithms [22, 38, 39], block-abstraction memoization [23, 24, 25, 83], program transformations [31, 33, 34, 41, 42], cooperative verification [16, 20], witness certification and validation [5, 19], and test-case generation [32, 52, 75]. The configurations of CPAchecker that were submitted to competitions are described in the competition contribution papers of SV-COMP [2, 3, 8, 55, 57, 66, 68, 69, 70, 74, 81, 82], Test-Comp [30, 60, 61, 62], and RERS [46, 47]. These publications give an indication of the breadth of analyses available in CPAchecker and its power and flexibility as a verification framework.

Questions, bug reports, and feature requests for CPAchecker are always welcome on its mailing list (https://groups.google.com/g/cpachecker-users) and the issue tracker (https://gitlab.com/sosy-lab/software/cpachecker/-/issues).

1.4 CPAchecker in Education

Due to the many algorithms and abstract domains, and the clean and extensible architecture, CPAchecker is an ideal tool for teaching of program-analysis techniques. The techniques can be explored in comparison and their effects observed. Visualizations of abstract states and error paths help understand the reasons for correctness or violation of the specification. We use CPAchecker in various courses on software engineering, software verification, software testing, and program semantics.

1.5 Outline

This tutorial starts in Sect. 2 with installation instructions and a first example of running CPAchecker. Section 3 explains the inputs and outputs of CPAchecker. Finally, Sect. 4 gives an overview on the most important analysis techniques that CPAchecker provides for software verification.

The extended version [7] includes further information on CPAchecker, provides an overview of all concrete example command lines together with references to the respective part of the tutorial, provides more information about the CPAchecker project, its development history, achievements, and licensing, provides some more detailed examples for the presented analysis techniques, and explains how to use CPAchecker for witness validation and test-case generation.

2 Getting Started with CPAchecker

In the following, we explain the installation and a few alternatives for executing CPAchecker on individual verification tasks.

For trying out CPAchecker and following this tutorial we provide a few example programs in a reproduction package [6]. We assume this package was downloaded and unpacked, and that the current working directory is its root directory (where directory `examples/` is visible). The execution of each example in this tutorial should take less than 10 seconds.

2.1 Local Installation

Installation Requirements. Most features of CPAchecker require a 64-bit GNU/Linux machine, unless users build the required libraries themselves. A limited feature set is usable on other platforms. We recommend a current LTS version of Ubuntu; recent versions of other distributions can be expected to work as well.

Installation. For users on Debian or Ubuntu we provide a package repository at `https://apt.sosy-lab.org`. Please follow the instructions on that webpage to enable the repository. Afterwards, the latest version of CPAchecker can be installed with `sudo apt install cpachecker`.

For users without root access or on other distributions, we also provide CPAchecker as pre-built binary releases via Zenodo [49] and our download page. Please ensure that a Java Runtime Environment (JRE) is available (for CPAchecker 3.0, Java version 17 or newer is required). Unpack the archive for CPAchecker after the download. We recommend adding CPAchecker's `bin/` directory to the `PATH` environment variable. This way the examples provided in this tutorial work as is, without having to specify the full path to the `cpachecker` executable every time. If CPAchecker was installed via the package repository, changing the `PATH` variable is not necessary.

Execution. To try out CPAchecker, run the following command from the reproduction package's [6] root directory:

```
cpachecker examples/example-safe.c
```

This will verify that there is no assertion violation in program `example-safe.c`, and report that the program satisfies the specification. Further information is provided in Sect. 2.4.

2.2 Execution via Container

CPAchecker is available as an image in OCI format, for use with container runtimes like Podman and Docker. The identifiers of the images are `sosylab/cpachecker` (always the latest release) and `sosylab/cpachecker:3.0` for version 3.0. The following command line executes CPAchecker 3.0 from a container (may require `sudo`, depending on the Docker installation):

```
docker run -v "$(pwd)":/workdir sosylab/cpachecker:3.0 \
    examples/example-safe.c
```

Command-line argument `-v "$(pwd)":/workdir` makes the current working directory (`$(pwd)`) available in the started container at path `/workdir`. This is the default entrypoint of the CPAchecker images. Command-line argument `-u $UID:$GID` might be added after `docker run` to set the user and group ID of the container to the current user and group ID: output files produced by CPAchecker are then owned by the current user instead of `root`. Argument `examples/example-safe.c` is passed to CPAchecker and will be explained in Sect. 2.4. The command-line arguments and input files can be adjusted as usual.

2.3 Remote Execution via Website

We provide a web interface for CPAchecker at `https://vcloud.sosy-lab.org/cpachecker/webclient/run/`. The examples of this paper are available as Examples on the left of the page.

2.4 Example Verification Task

For all example command lines in this paper we assume a local installation of CPAchecker and that the artifact with the examples [6] has been unpacked in the current directory (such that the directory `examples/` is present). If necessary, e.g., for Docker usage, please adjust the command lines accordingly.

Program Description. We use the program in Fig. 2a. This program initializes variables `n` and `x` to two nondeterministic but concrete values of type `unsigned int` (modeled by calls to `__VERIFIER_nondet_uint()`) and then initializes `y` to the difference between `n` and `x`. As long as `x` is larger than `y`, the `while` loop decrements `x` and increments `y` by one. If the sum of `x` and `y` does not equal `n` at the end of a loop iteration, `__assert_fail` at line 10 triggers a program error (arguments omitted for simplicity). The program is correct with respect to the specification that `__assert_fail` is unreachable, because the sum of `x` and `y` always equals `n` at the end of every loop iteration. A variant of this program is shown in Fig. 2b. The variant follows the same execution except at line 9. Here an error is triggered if `x` is smaller than `y`. This error is reachable by initializing `n` to 3 and `x` to 2 (among many other possibilities).

Verification Run. To verify the example program in Fig. 2a with CPAchecker, execute the below command in a terminal (cf. example default on the web service):

```
cpachecker examples/example-safe.c
```

This command line does not specify an explicit configuration. In this case CPAchecker uses the default configuration, which is the currently recommended

```
1  extern unsigned
       __VERIFIER_nondet_uint();
2  extern void __assert_fail();
3  int main() {
4    unsigned n =
         __VERIFIER_nondet_uint();
5    unsigned x =
         __VERIFIER_nondet_uint();
6    unsigned y = n - x;
7    while(x > y) {
8      x--; y++;
9      if (x + y != n) {
10       __assert_fail();
11     }
12   }
13   return 0;
14 }
```

(a) `example-safe.c` (error unreachable)

```
1  extern unsigned
       __VERIFIER_nondet_uint();
2  extern void __assert_fail();
3  int main() {
4    unsigned n =
         __VERIFIER_nondet_uint();
5    unsigned x =
         __VERIFIER_nondet_uint();
6    unsigned y = n - x;
7    while(x > y) {
8      x--; y++;
9      if (x < y) {
10       __assert_fail();
11     }
12   }
13   return 0;
14 }
```

(b) `example-unsafe.c` (error reachable)

Fig. 2: Example C programs

configuration. Like most configurations shipped with CPAchecker, the default configuration uses the default specification, which specifies that no C assertion error `__assert_fail` and no label named `ERROR` should be reachable. The specifications, configurations, and the available analyses are described in more detail in Sect. 3.2, Sect. 3.3, and Sect. 4.

At the end of its execution, CPAchecker produces the following messages:

```
Verification result: TRUE. No property violation found by chosen configuration.
More details about the verification run can be found in the directory "./output".
Graphical representation included in the file "./output/Report.html".
```

The verification result `TRUE` indicates that the error (line 10 in Fig. 2a) is not reachable. We can also change the input program to `example-unsafe.c` in the command line. In this case, the verification result is `FALSE`, meaning that CPAchecker finds an execution path that triggers the error. The meanings of verification results and how to navigate through the generated report is the topic of Sect. 3.4 and Sect. 3.5, respectively.

3 Input and Output Interface of CPAchecker

Figure 1 shows the inputs and outputs of CPAchecker. CPAchecker always takes a program, a specification, and a configuration as input. It always produces a verdict and a report. Depending on how the user intends to use it, either as a verifier, a witness validator, or a test-case generator, CPAchecker may also take a verification witness as input, or produce witnesses or test cases as output.

3.1 Input Program

CPAchecker supports a large subset of the GNU-C11 features. Normally, the verifier expects pre-processed input files. CPAchecker supports compiler directives (e.g., `#include` or `#define`) if the command-line argument `--preprocess`

Table 1: Provided specifications (files in `config/specification/`)

Specification	Description
`ErrorLabel`	Labels named `ERROR` (case insensitive) are never reachable.
`Assertion`	All `assert` statements hold.
`default`	Both `ErrorLabel` and `Assertion` hold.
`overflow`	All operations with a signed-integer type never produce values outside the range representable by the respective type.
`datarace`	Concurrent accesses to the same memory location must be atomic if at least one of them is a write access.
`memorysafety`	All memory deallocations and pointer dereferences are valid and all allocated memory is pointed to or deallocated when the program exits.
`memorycleanup`	All allocated memory is deallocated before the program exits.

is given, in which case CPAchecker pre-processes the input C program. To guarantee a meaningful verification of programs that use external functions, including functions in the C standard library, the implementations of the functions have to be provided in the input programs. Otherwise, CPAchecker overapproximates their behavior, potentially leading to false alarms. Two exceptions are the function `pthread_create` for creating a new thread and functions `malloc`, `memset`, etc., for manipulating memory, which are handled out-of-the-box by CPAchecker's concurrency and memory analyses, respectively. To verify a software project that consists of multiple C files, all relevant files must be listed on the command-line. By default, CPAchecker starts the analysis from the function `main`. Another entry function can be specified with the command-line argument `--entry-function <entry function>`.

The semantics of a C program depends on the runtime platform, which consists of a machine architecture, a data model, and an operating system. CPAchecker assumes a single platform during verification. The command-line argument `--32` (default) sets the platform to 32-bit x86 Linux (ILP32) and `--64` sets the platform to 64-bit x86 Linux (LP64) [78].

3.2 Program Specification

Besides the input program, a *specification* is needed as input for CPAchecker. The specification defines what property of the program should be checked. CPAchecker supports an automaton-based specification language (similar to Blast [17] and Slam [9]) to define program specifications (documented in `doc/SpecificationAutomata.md`). CPAchecker ships with several common specifications in the directory `config/specification/`. A selection is listed in Table 1. CPAchecker also supports property files written in the specification language that was standardized by the International Competition on Software Verification (SV-COMP) [13].

The command-line argument `--spec <specification>` defines the specification to use. It accepts the path to a specification-automaton file, an SV-COMP property file, or the name of one of the specifications that ship with CPAchecker. For

```
1 OBSERVER AUTOMATON AssertionErrorAutomaton
2 INITIAL STATE Init;
3 STATE USEFIRST Init :
4   // AST-based matching of function calls to __assert_fail
5   MATCH {__assert_fail($?)}
6     -> ERROR("assertion in $location");
7 END AUTOMATON
```

Fig. 3: Example of automaton-based specification for checking assert statements

example, to verify a program against the provided specification `Assertion` with CPAchecker's default analysis, we run (cf. example assert on the web service):

```
cpachecker [--preprocess] --spec Assertion examples/example-safe.c
```

The square brackets in the above command indicate that argument `--preprocess` may be omitted if the program does not contain compiler directives (cf. Sect. 3.1).

Figure 3 shows a simplified version of the `Assertion` specification. The specification is violated if a call to function `__assert_fail` is reachable in the given input program, which matches how `assert` statements appear in a C program after pre-processing. The automaton starts in the initial state `Init` and observes the analyzed program operations until an operation matches a call to `__assert_fail` (line 5) with an arbitrary number of function-call arguments (denoted by `$?`). In this case, the automaton transitions to the special state `ERROR` (line 6) that signals a specification violation with the given explanation.

3.3 CPAchecker Configuration

CPAchecker is highly configurable via a set of configuration options, which are documented in the file `doc/ConfigurationOptions.txt`. Configuration options are specified as key-value pairs in a configuration file or on the command line. An extensive set of bundled configuration files is available in directory `config/`. Most of these bundled configurations specify default values for common configuration options, e.g., the specification `config/specification/default.spc` and a time limit of 900 s. Command-line arguments overwrite these defaults.

It is possible to write and provide own configuration files. Their format is inspired by Windows INI files with some extensions like include directives. A full description is available in `doc/Configuration.md`. Configuration files may use relative paths. CPAchecker interprets these relative paths relative to the directory of the respective configuration file.

Command-line argument `--config CONFIG_FILE` selects a configuration file. The bundled configuration files can also be selected with short-hand arguments that consist of the base name of the configuration file, e.g., `--kInduction` for the configuration file `config/kInduction.properties` or `--svcomp24` for the configuration file `config/svcomp24.properties`. When no configuration file is explicitly specified, CPAchecker runs in its default configuration (defined by the configuration file `config/default.properties`).

The command-line argument `--option key=value` sets a single configuration option. The order of command-line arguments is irrelevant. If an option is set both in the configuration file and through `--option`, the `--option` value takes precedence and overwrites any value from the configuration file.

CPAchecker provides shortcuts for the most common configuration options, for example `--64` to specify the platform as 64-bit x86 Linux (LP64), or `--timelimit` to set an analysis time limit. A full list of shortcuts is available via `cpachecker -h` and in `doc/Configuration.md`. For technical reasons, a few command-line arguments exist that can only be specified through command-line arguments and not via configuration files. These arguments include `--benchmark` (which leads to better performance by disabling CPAchecker-internal assertions, writing no output files, and much more) and `--heap` (which adjusts the amount of memory used by the JVM for CPAchecker).

As an example, consider the following command line (cf. example settingOptions on the web service):

```
cpachecker --kInduction --timelimit 900s --heap 2000M \
    --spec ErrorLabel examples/example-safe.c \
    --option solver.solver=MATHSAT5
```

This invokes CPAchecker with the configuration for k-induction, sets the configuration option `limits.time.cpu` for the time limit to 900 s, tells the JVM to use 2 000 MiB of heap memory, chooses the specification file `ErrorLabel`, the program `program.c` as input file, and sets the configuration option `solver.solver` to `MATHSAT5`.

3.4 Verification Verdict

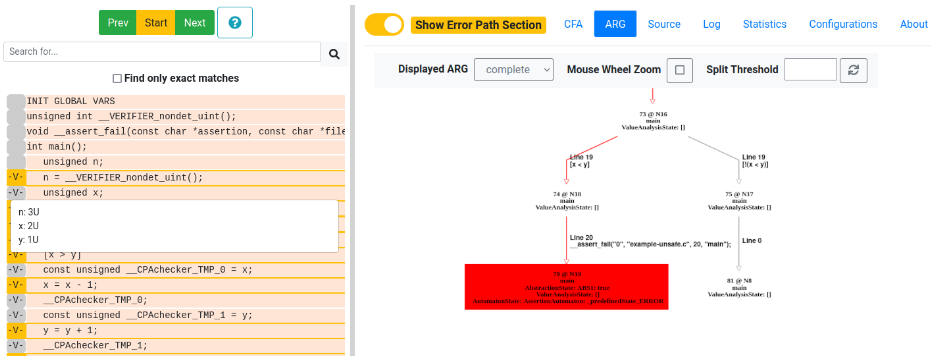
CPAchecker may report three different verification verdicts: (1) `TRUE`, if it proves that the program *satisfies* the specification; (2) `FALSE`, if it proves that the program does *not satisfy* the specification; (3) `UNKNOWN`, if it cannot decide the verification task using the given resource limits and configuration.

3.5 Interactive Report in HTML Format

In addition to a verification verdict, CPAchecker produces detailed information about the performed analysis in directory `output/` in the current working directory. This usually includes an interactive report in HTML format. Note that different configurations may produce different output files.

The interactive report offers a graphical interface for users to inspect the results of CPAchecker. It allows to inspect, among others: the *control-flow automata* (CFA) of the input program, the *abstract reachability graph* (ARG) that was constructed by the chosen configuration, statistics, and an error path that violates the specification (if the verdict is `FALSE`).

In the following we explain the most important parts of this report. A screenshot of the report is shown in Fig. 4. An example report is provided online. If



```
    INIT GLOBAL VARS
    unsigned int __VERIFIER_nondet_uint();
    void __assert_fail(const char *assertion, const char *file
    int main();
       unsigned n;
-V-    n = __VERIFIER_nondet_uint();
-V-    unsigned x;
n: 3U
x: 2U
y: 1U
-V-    [x > y]
-V-    const unsigned __CPAchecker_TMP_0 = x;
-V-    x = x - 1;
-V-    __CPAchecker_TMP_0;
-V-    const unsigned __CPAchecker_TMP_1 = y;
-V-    y = y + 1;
-V-    __CPAchecker_TMP_1;
```

```
73 @ N16
main
ValueAnalysisState: []

Line 19
[x < y]

Line 19
[!(x < y)]

74 @ N18
main
ValueAnalysisState: []

75 @ N17
main
ValueAnalysisState: []

Line 20
__assert_fail("0", "example-unsafe.c", 20, "main");

Line 0

79 @ N19
main
AbstractionState: ABS1: true
ValueAnalysisState: []
AutomatonState: AssertionAutomaton: _predefinedState_ERROR

81 @ N8
main
ValueAnalysisState: []
```

Fig. 4: Screenshot of the HTML report for program `example-unsafe.c`

CPAchecker reports the verdict `FALSE` and the used analysis provides detailed counterexample information, the report file is `output/Counterexample.0.html` (number `0` may differ). Otherwise, the report file is `output/Report.html`.

Control-Flow Automata. The tab CFA in the report shows the input program in the internal representation of CPAchecker, the control-flow automata (CFA). A CFA consists of program locations (nodes of the graph) and program statements (edges of the graph). In the report, a double-click on a CFA edge navigates to the source-code line it represents. The drop-down menu "Displayed CFA" can be used to display a single CFA for a single program function.

Abstract-Reachability Graph. The tab ARG in the report shows a graphical representation of the program states that were explored by CPAchecker in the form of an abstract-reachability graph (ARG). The right-hand side of Fig. 4 shows an ARG. Each node in the ARG represents an *abstract* state of the input program. CPAchecker constructs abstract states according to the selected configuration. An abstract state usually represents a set of *concrete* program states in order to overapproximate the reachable state space. Two abstract states are connected by a directed edge if one state is the successor to the other. The directed edge goes from predecessor to successor and is labeled with a program operation that induced the predecessor-successor relation during analysis.

If CPAchecker reported the verdict `TRUE`, the ARG represents all reachable abstract program states. If CPAchecker reported the verdict `FALSE`, nodes and edges that are part of the error path are marked in red (as in Fig. 4).

Error Path. If the verification verdict is `FALSE` and the analysis provides detailed counterexample information, the report includes a textual error-path section as separate panel on the left (toggle with button "Show Error-Path Section"). This allows users to step through the error path that CPAchecker computed. The textual error path is a list of program statements, accompanied by concrete assignments to all variables on the error path. A button -V- is displayed next to each statement, which indicates the concrete variable assignments at the respective location. To replay the error path step-by-step, users can click on

```
<...>
content:
- invariant:
    type: "loop_invariant"
    location:
      file_name: "example-safe.c"
      line: 7
      column: 3
      function: "main"
    value: "( x + y == n )"
    format: "c_expression"
```

(a) Relevant sections of a correctness witness for the safe program in Fig. 2a

```
<...>
content:
 - segment:
   - waypoint:
       type: assumption
       location:
         file_name: "example-unsafe.c"
         line: 6
       constraint:
         value: "x == 0 && n == 1"
 - segment:
   - waypoint:
       type: target
       location:
         file_name: "example-unsafe.c"
         line: 10
```

(b) Relevant sections of a violation witness for the unsafe program in Fig. 2b

Fig. 5: Example verification witnesses (format version 2.0, slightly shortened for readability)

the Start button on the top left. Then, two buttons Next and Prev can be used to navigate through the error path.

3.6 Statistics

CPAchecker collects a variety of statistics, depending on the chosen analysis. These are presented in the interactive report under tab Statistics and are also written to file `output/Statistics.txt`. With the command-line argument `--stats`, CPAchecker prints the statistics to the console at the end of the verification run.

The statistics help users to evaluate the performance of the analysis. Below is an example excerpt of a run's statistics that shows the time spent on SMT solving, the total number of computed reachable abstract states, and the consumed CPU time.

```
Total time for SMT solver (w/o itp):     0.017s
[...]
Size of reached set:             10
[...]
CPU time for analysis:            0.860s
```

A separate tutorial covers how to interpret CPAchecker statistics in more detail.

3.7 Verification Witnesses

Verification witnesses [5, 19] help users and tools to reason about verification results and allow independent validation of the verification result. Validation is usually easier than verification, thanks to the additional information the witness provides. CPAchecker can both export witnesses for verification results and validate witnesses that other tools produce. The extended version [7] explains witness validation in detail.

Correctness Witnesses. Correctness witnesses are defined for reachability of error locations and detection of signed-integer overflows in sequential programs.

```
 1 unsigned __VERIFIER_nondet_uint() {
 2   static unsigned call_count = 0;
 3   unsigned retval;
 4   switch (call_count) {
 5     case 0: retval = 2U; break;
 6     case 1: retval = 2U; break;
 7   }
 8   ++call_count;
 9   return retval;
10 }
```

Fig. 6: Test harness generated for the example program in Fig. 2b

CPAchecker produces such a witness not only if the verdict is `TRUE`, but also if it is `UNKNOWN` (in this case with partial information). The witness contains information about the explored program state space in the form of loop and location invariants. In case the analysis result is `TRUE`, the invariants hold whenever the program execution passes through the respective location.

Figure 5a shows an excerpt of a correctness witness for the safe program in Fig. 2a. It reports the loop invariant `x + y == n` for the loop head in line 7.

Violation Witnesses. Violation witnesses represent one or more program paths that lead to a specification violation. This is achieved by specifying assumptions about the program inputs and the control flow of the program.

Figure 5b shows an excerpt of a violation witness for the unsafe program in Fig. 2b. It shows the program path that leads to the assertion failure at line 10 when `x` is assigned value 0 and `n` is assigned value 1.

3.8 Test Harnesses

If CPAchecker finds a specification violation (verdict `FALSE`), it produces a test harness that triggers this violation through test execution. A test harness contains a sequence of external inputs (e.g., for inputs modeled by `__VERIFIER_nondet*`) to the program that trigger an execution path to the specification violation. Figure 6 shows an excerpt of a test harness for the example program in Fig. 2b. The two return values `2U` (lines 5 and 6) initialize, in the program under analysis (Fig. 2b), both variables `n` and `x` with value 2. This triggers the assertion failure at line 10 of the program.

The test harness can be compiled with the program under analysis:

```
gcc output/Counterexample.1.harness.c examples/example-unsafe.c
```

This produces a binary `a.out`. The execution of `./a.out` exhibits that the claimed specification violation is actually reachable. It reports:

```
CPAchecker test harness: property violation reached
```

The extended version [7] gives more details on test generation with CPAchecker.

Table 2: Commonly-used configurations and supported specifications

Configuration	Specification (cf. Sect. 3.2)	Description
Configurations for reachability specifications:		
`--valueAnalysis-NoCegar-join`	`default`, `Assertion`, `ErrorLabel`, custom automaton specifications, and SV-COMP property `unreach-call.prp`	Section 4.2
`--symbolicExecution-NoCegar`		Section 4.4
`--predicateAnalysis`		Section 4.5
`--bmc-incremental`		Section 4.6
`--kInduction`		Section 4.7
Special-purpose configurations:		
`--smg`	memory safety (`memorysafety` and `memorycleanup`)	Section 4.8
`--lassoRankerAnalysis`	termination	Section 4.9
`--terminationToSafety`		
`--predicateAnalysis--overflow`	`overflow`	Section 4.10
`--dataRaceAnalysis`	`datarace`	Section 4
Meta configurations:		
`--svcomp24`	reachability specifications	[8]
default (no argument)	and all SV-COMP properties	Section 4.1

4 Verification Analyses and How to Select Them

This section shows how to execute various commonly-used verification analyses in CPAchecker. These analyses can be divided into three groups depending on the kind of specifications they can check. First, there are analyses that perform a reachability analysis. These support common specifications, for example, reachability of an error location or an assertion violation. Second, there are analyses that support a particular special-purpose specification. Third, there are meta analyses that implement strategy selection and delegate to one of the above depending on the provided specification. Table 2 lists common configurations and the respective specifications they support. Apart from the configuration `--dataRaceAnalysis`, which performs partial order reduction [73] over memory accesses in combination with value analysis [43], the following sections explain these configurations in more detail.

4.1 Selecting an Analysis

Selecting an analysis of CPAchecker primarily depends on the kind of specification that should be verified. Memory safety, overflows, and data races can each be verified by exactly one recommended analysis, which is listed in Table 2. For termination, there are two recommendations, described in Sect. 4.9. If SV-COMP property files are used to encode the specification, meta configurations of CPAchecker automatically select a recommended analysis depending on the specification.

For standard reachability specifications a wide range of different analyses and techniques is available in CPAchecker. Each of them has their strengths and weaknesses, and while some of them are more powerful or efficient in general,

Table 3: Main configuration flavors of value analysis

Precision Refinement	Path Sensitivity	Configuration
✗	✗	`--valueAnalysis-NoCegar-join`
✗	✓	`--valueAnalysis-NoCegar`
✓	✓	`--valueAnalysis-Cegar`

none of them always outperforms all of the others, so it can be worthwhile experimenting with several analyses.

The general recommendation for most use cases is the default analysis of CPAchecker (used if no other configuration is selected on the command line). It is a meta configuration that uses k-induction (`--kInduction`, most effective overall in our experience) for reachability specifications.

CPAchecker's value analysis (`--valueAnalysis-NoCegar-join`), symbolic execution (`--symbolicExecution-NoCegar`), and bounded model checking (BMC, `--bmc-incremental`) are mostly suited for finding specification violations. While they are often quite efficient in finding bugs, they are often inefficient for proving correctness for large programs. In our experience these configurations usually either succeed quickly or will not produce a result at all.

To prove the absence of specification violations in larger programs, either abstraction of the program state space or a proof technique such as induction needs to be used. Value analysis and symbolic execution support a limited form of abstraction (ignoring irrelevant program variables and clauses) if their configuration variants with precision refinement are chosen as described in the respective sections below. Predicate abstraction (`--predicateAnalysis`) is stronger and can in principle find arbitrary loop invariants as long as the loop invariants do not require quantifiers nor floating-point arithmetic. k-Induction (`--kInduction`) on the other hand requires that an induction proof can be found for the program.

Another aspect that needs to be considered is that value analysis and symbolic execution in CPAchecker do not support precise reasoning about dynamically allocated memory and data structures on the heap, whereas BMC, predicate abstraction, and k-induction do support this. However, the latter three are based on solving (sometimes large) formulas with an SMT solver, which may not scale. Value analysis has the advantage that it does not require SMT solving, but the disadvantage that it cannot reason about non-deterministic values. Symbolic execution uses an SMT solver, but only when required for non-deterministic values.

The value analysis can be considered comparatively easy to understand conceptually, which makes it a good starting point for the use of CPAchecker.

4.2 Value Analysis

CPAchecker's value analysis tracks concrete value assignments. There are two main configuration choices for the value analysis: (1) whether to use precision refinement, and (2) whether to be path sensitive. Table 3 lists the available

```c
extern void __assert_fail();
int main() {
  int x = 0;
  int y = 0;
  int z = 0;
  while (x < 2) {
    x++;
    y = z + 1;
  }
  if (z != 0) {
    __assert_fail();
  }
  return 0;
}
```

Fig. 7: Program example-const.c

```c
extern unsigned __VERIFIER_nondet_uint();
extern void __assert_fail();
int main() {
  unsigned int x = __VERIFIER_nondet_uint();
  unsigned int y = x;
  unsigned int z = __VERIFIER_nondet_uint();
  while (x < 2) {
    x++;
    y++;
    z = x + z;
  }
  if (x != y) {
    __assert_fail();
  }
  return 0;
}
```

Fig. 8: Program example-sym.c

command-line arguments to run CPAchecker with the corresponding configuration of value analysis. For example, the following command runs a configuration of value analysis that implements constant propagation [1] (no precision refinement, no path sensitivity) on the program in Fig. 7 (cf. example valueAnalysis-NoCegar-join on the web service):

```
cpachecker --valueAnalysis-NoCegar-join examples/example-const.c
```

This configuration tracks only value assignments that always hold on a given location, because abstract states are joined when control flow meets. This is efficient, but in most cases not powerful enough to verify programs. For Fig. 7, it suffices because only the value of variable `z` is needed to prove the program safe, and this is always 0. The extended version [7] shows the state-space exploration of the value analysis for this example in more detail. If, however, the program safety would also depend on the values of `x` or `y` after the loop, the verification result would be `UNKNOWN` because the analysis does not track these non-constant variable values.

The value analysis with path sensitivity tracks value assignments per program path and location. For the example in Fig. 7, it would keep track of all variable values and fully unroll the loop. This leads to path explosion when many paths with distinct value assignments exist, because the analysis tracks all of them separately.

Value analysis with path sensitivity and precision refinement mitigates this path explosion by tracking only those value assignments that are necessary for the analysis to prove the program safe. This is more efficient than value analysis without precision refinement in the common case where not all variables in the program are relevant for safety, like in Fig. 7. The relevant variables are detected automatically through counterexample-guided abstraction refinement (CEGAR) with Craig interpolation [43].

Because the value analysis always tracks concrete value assignments and overapproximates nondeterministic values, it may find false alarms. To mitigate this, CPAchecker runs a precise, SMT-based feasibility check on every found potential error path and only reports confirmed specification violations. This can be seen in the output of CPAchecker, which is provided in the extended version [7].

4.3 Interval-Based Data-Flow Analysis

The data-flow analysis (DF) of CPAchecker is a lightweight proof-finding technique that uses *arithmetic expressions over intervals* as its abstract domain [15, 21]. It tracks, for an automatically-selected set of program variables, the range of values that each variable can take in the form of interval expressions, e.g., $[l_1, u_1] \cup [l_2, u_2]$, where l_i (resp. u_i) is a numerical value representing the lower (resp. upper) bound of an interval. DF supports dynamic precision refinement. At the beginning of the analysis, it performs a coarse but efficient program exploration. If some abstract state reachable in the exploration violates the safety specification, DF incrementally increases its precision by tracking more program variables, allowing more complex expressions of intervals, and disabling widening [54]. To run DF in CPAchecker, provide the configuration `--dataFlowAnalysis` on the command line (cf. example dataFlowAnalysis on the web service):

```
cpachecker --dataFlowAnalysis examples/example-const.c
```

For the above example, CPAchecker produces the verdict `TRUE`. A limitation of DF is that its abstract program exploration cannot identify concrete error paths when there are specification violations and may sometimes be too imprecise to find a safety proof. For example, when CPAchecker analyzes `example-safe.c` or `example-unsafe.c` in Fig. 2 with DF, it produces the verdict `UNKNOWN`. DF cannot only run standalone but also serve as an auxiliary invariant generator that assists other analyses, e.g., k-induction [20] (cf. Sect. 4.7).

4.4 Symbolic Execution

The symbolic execution [40] of CPAchecker tracks concrete value assignments the same way as the value analysis. But for every value that cannot be tracked concretely, for example because it is assigned non-deterministically, symbolic execution introduces a new symbolic value s_i. Whenever a symbolic value is used in an expression, symbolic execution stores the expression over this symbolic value without evaluating it. In addition, symbolic execution tracks the constraints over these symbolic values for each program path. This produces a symbolic-execution tree (cf. the extended version [7] for details). From this, concrete variable assignments can be derived for any program path. The symbolic execution of CPAchecker also supports precision refinement through CEGAR with Craig interpolation [39]. This determines which variables and constraints must be tracked through the program.

The below command runs a configuration of symbolic execution [65] without precision refinement (cf. example symbolicExecution-NoCegar on the web service):

```
cpachecker --symbolicExecution-NoCegar examples/example-sym.c
```

Because symbolic execution tracks the expressions over symbolic values without further abstraction, it is well suited for collecting constraints on inputs for certain program paths. But this precision also leads to path explosion: The analysis of symbolic execution on program `example-safe.c` (Fig. 2a) does not terminate. To prove the program safe, it is important to know that the sum of x and y equals n at line 9. Symbolic execution tracks this by storing the expressions $n = s_1, x = s_2, y = s_1 - s_2$, $x = s_2 - 1, y = s_1 - s_2 + 1$, $x = s_2 - 1 - 1, y = s_1 - s_2 + 1 + 1$, and so on. This produces ever more complicated expressions and does not scale.

The following command runs a configuration of symbolic execution with precision refinement (cf. example symbolicExecution-Cegar on the web service):

```
cpachecker --symbolicExecution-Cegar examples/example-sym.c
```

On the program of Fig. 8, this only tracks assignments and constraints over x and y, which are necessary to prove the program safe. Assignments to z are not tracked.

4.5 Predicate Abstraction

Predicate abstraction [36, 59, 63] abstracts the program's state space with predicates that it learns using CEGAR [53] and Craig interpolation [59]. Compared to symbolic execution, predicate abstraction is not limited to tracking (symbolic) values and constraints in the program, but can derive more powerful abstractions. The computation of abstractions can be costly, thus predicate abstraction uses *large-block encoding* [18, 36] to compute abstractions only at certain program locations, which by default are the loop-head locations. This reduces the number of abstractions calculated and, hence, the overall cost. To run predicate abstraction, use the command (cf. example predicateAnalysis on the web service):

```
cpachecker --predicateAnalysis examples/example-safe.c
```

In this example, predicate abstraction derives the loop invariant `x + y == n`, which proves that `__assert_fail` in Fig. 2a is unreachable, and hence returns the verdict `TRUE`. Learned predicates at these locations are written down in a format based on SMT-LIB2 [10] into the file `output/predmap.txt` of the current working directory. Take the program in Fig. 2a for example. Predicate abstraction can derive the invariant `x + y == n` for the `while` loop at line 7 in function `main` that suffices to prove the safety specification that the assertion error is unreachable. In `predmap.txt`, this is represented as follows:

```
(declare-fun |main::n| () (_ BitVec 32))
(declare-fun |main::y| () (_ BitVec 32))
(declare-fun |main::x| () (_ BitVec 32))

main:
(assert (= |main::n| (bvadd |main::y| |main::x|)))
```

Predicate abstraction can abstract the program state space concisely in a way that proves the program safe, if it learns the right predicates. Unfortunately, there is no mechanism forcing predicate abstraction to find predicates that abstract well. Especially for concrete value assignments in the program, the learned predicates might enumerate all possible states. For instance, predicate abstraction may unnecessarily learn the predicates `x == 0`, `x == 1`, and `x == 2` at line 6 of Fig. 7, instead of `z == 0`. Alternatively, Impact [72] is another analysis that abstracts a program's state space with predicates. It computes and refines abstractions in a lazier way compared to predicate abstraction, and can be initiated using the configuration `--predicateAnalysis-ImpactRefiner-ABEl`. The two analyses have shown different and complementing strengths in our empirical evaluations [22]: Predicate abstraction is more effective at deriving proofs, whereas Impact is more efficient at finding specification violations.

4.6 Bounded Model Checking

Bounded model checking (BMC) [22, 51] is an analysis specialized in finding specification violations. Given a bound n, BMC symbolically unrolls the loops in the program n times, encodes all execution paths and specification violations (within the unrolling bound n) into an SMT formula, and checks the satisfiability of the formula with an SMT solver. The satisfiability of the formula directly corresponds to the feasibility of the encoded error paths. If the formula is satisfiable, then a specification-violating execution path (with n loop unrollings) exists and can be extracted from the satisfying assignment. A bounded model checker then reports the verification verdict `FALSE`. In case the formula is unsatisfiable, the program is considered safe up to the bound n. A bounded model checker reports the verification verdict `TRUE` if the loops in the program have finite bounds and are fully unrolled by the bound n. Otherwise, the verdict is `UNKNOWN`, as the behavior of the program at higher unrolling bounds is still unknown.

CPAchecker automatically determines the required unrolling bound by incrementally increasing the bound using configuration `--bmc-incremental`. Incremental BMC starts with an unrolling bound of 0 and increments the bound by 1 after each iteration. The analysis terminates once an error path is found, the safety specification is proven (by fully unrolling all loops in the program), or a resource limit is reached. For instance, the following command runs BMC with incrementally increasing loop bound on the program in Fig. 2b (cf. example bmc-unsafe on the web service):

```
cpachecker --bmc-incremental examples/example-unsafe.c
```

CPAchecker finds the bug inside the loop body of the program in Fig. 2b on its first encounter of the assertion, with zero complete unrollings of the loop. Running incremental BMC on the correct program in Fig. 2a does not succeed (cf. example bmc-safe on the web service). During the process, CPAchecker produces log messages that show the current unrolling bound:

```
Adjusting maxLoopIterations to 2
↪  (LoopBoundCPA:LoopBoundPrecisionAdjustment.nextState, INFO)
```

CPAchecker eventually reaches the time limit and the verdict is `UNKNOWN`, since a really large unrolling bound (roughly 2^{31}) is required to fully explore the program. If the loop condition at line 7 changes to `x > 0 && x < 3` in Fig. 2a, incremental BMC can prove the program safe with 2 complete loop unrollings.

4.7 Extensions of BMC for Unbounded Verification

BMC can be extended for unbounded verification of programs by employing the k-induction principle [20, 77] or constructing fixed points, i.e., inductive invariants, via Craig interpolation [38, 71, 79, 80]. To run k-induction in CPAchecker, use the configuration `--kInduction`, which combines k-induction with an auxiliary invariant generator based on data-flow analysis [15, 20] (described in Sect. 4.3). The invariants produced by the latter are used to strengthen the induction hypotheses of the former. This is more effective than plain k-induction [20]. As opposed to incremental BMC, k-induction could easily prove the safety of the example programs in Fig. 2a with the command (cf. example kInduction on the web service):

```
cpachecker --kInduction examples/example-safe.c
```

CPAchecker has three verification algorithms based on BMC and Craig interpolation: interpolation-based model checking (IMC) [38, 71], interpolation-sequence-based model checking (ISMC) [14, 79], and dual approximated reachability (DAR) [14, 80]. From unsatisfiable BMC queries, the three algorithms derive interpolants to construct inductive invariants at loop heads. Such an invariant overapproximates the reachable states of the program that conforms to the safety specification, and hence could serve as a proof for the program's correctness. IMC, ISMC, and DAR are enabled via the configurations `--bmc-interpolation`, `--bmc-interpolationSequence`, and `--bmc-interpolationDualSequence`, respectively, and currently support only programs with at most one loop. The tool CPAchecker verifies the program in Fig. 2a with IMC (`--bmc-interpolation`) via the command (cf. example bmc-interpolation on the web service):

```
cpachecker --bmc-interpolation examples/example-safe.c
```

It produces the below log message:

```
The current image reaches a fixed point
↪  (IMCAlgorithm.reachFixedPointByInterpolation, INFO)
```

The message indicates that IMC has found an inductive invariant for the `while` loop at line 7 and proved the safety specification of the program.

4.8 Symbolic Memory Graphs with Symbolic Execution

CPAchecker's symbolic-memory-graph (SMG) analysis [56] combines symbolic execution [65] with a graph-based domain that tracks all memory. It is usable in CPAchecker with the configuration `--smg`. In addition to common state-space exploration, the SMG analysis can check for memory safety. The analysis can detect memory leaks, invalid memory access, and invalid freeing of memory.

SMGs accurately track most memory operations, including pointer arithmetics and bit-precise reading of memory. They also store memory boundaries and can thus be used to reason about the validity of pointer dereferences. A distinguishing feature of SMGs is that linked lists of arbitrary length can be abstracted under

```
 1  #include <stdlib.h>
 2  #include <assert.h>
 3  extern int __VERIFIER_nondet_int();
 4  int main() {
 5    int size = 100;
 6    int num = __VERIFIER_nondet_int();
 7    int * arr = malloc(sizeof(int) * size);
 8    for (int i = 0; i < size; i++) {
 9      arr[i] = num;
10      num++;
11    }
12    for (int i = size; i >= 0; i--) {
13      assert(*(arr + i) == num);
14      num--;
15    }
16    return 0;
17  }
```

Fig. 9: `example-unsafe-memsafety.c` with two distinct memory-safety violations

certain circumstances. This is currently limited to lists that terminate in indefinitely repeating equal values. If the analysis fails to abstract lists of arbitrary length, it enumerates all possible list lengths. This may lead to a path explosion, but can still find violations to safety specification.

We can see some capabilities of the SMG analysis on the example program in Fig. 9. The program first allocates some memory at line 7, then uses this memory to store some distinct but non-deterministic values in a loop at line 9, filling the entire memory allocated in `arr`. Then, in a reversed loop, the saved values are compared to their expected values at line 13. Please note that this example is not pre-processed and thus the command-line argument `--preprocess` is needed. To start the verification of memory safety with the configuration `--smg` on this program, run the following command:

```
cpachecker --preprocess --smg --spec memorysafety \
    examples/example-unsafe-memsafety.c
```

This detects that the first memory access of the second loop at line 12 is unsafe (i.e., the verdict is `FALSE`), as the pointer dereference exceeds the bounds of the allocated memory. Another error can be found before line 16, as the memory allocated in `arr` is never freed. This second memory-safety violation can be found either by fixing the invalid dereference at line 13, or by using the dedicated specification `memorycleanup`:

```
cpachecker --preprocess --smg --spec memorycleanup \
    examples/example-unsafe-memsafety.c
```

4.9 Termination Analysis

The specification *termination* requires a program to always terminate. A program that can execute infinitely is called *non-terminating*.

CPAchecker provides two approaches for termination analysis: the termination-as-safety analysis [76] `--terminationToSafety` and the lasso-based analysis [58] `--lassoRankerAnalysis`. Analysis `--terminationToSafety` is based on loop unrolling (similar to BMC, cf. Sect. 4.6). It can prove termination only if all loops

```
1   extern unsigned
         __VERIFIER_nondet_uint();
2   int main() {
3     unsigned int n = 1;
4     unsigned int z =
          __VERIFIER_nondet_uint();
5     while (n <= z) {
6       n = n + 1;
7       z = z - 1;
8     }
9     return 0;
10  }
```

(a) example-terminating.c

```
1   extern unsigned
         __VERIFIER_nondet_uint();
2   int main() {
3     int n = 1;
4     int z =
          __VERIFIER_nondet_uint();
5     while (n <= z) {
6       n = (n - 1) % 3;
7       z = (z + 1) % 3;
8     }
9     return 0;
10  }
```

(b) example-nonterminating.c

Fig. 10: Example C programs for demonstration of termination analyses

in the program can be fully unrolled, but is often efficient in finding specification violations, i.e., counterexamples that show non-termination. Analysis `--lassoRankerAnalysis` constructs ranking functions and does not need to unroll all loops in the program for termination proofs.

Termination-as-Safety Analysis. The termination-as-safety analysis transforms a verification task for a termination specification into a verification task for reachability. It stores the values of variables that were seen at the programs' loop heads. For example, the loop head for the two programs in Fig. 10 is the location that corresponds to line 5. Similar to BMC (cf. Sect. 4.6), when the analysis visits a loop head for the $n + 1$-st time, it constructs an SMT formula that symbolically represents n loop unrollings. Via satisfiability queries, the analysis checks whether there exists a reachable state that is visited twice within n loop iterations. If such a state is found, the program is non-terminating.

The following command line runs the analysis on the program in Fig. 10b (cf. example terminationToSafety on the web service):

```
cpachecker --terminationToSafety examples/example-nonterminating.c
```

CPAchecker reports the verdict `FALSE` and produces a counterexample that shows the following three unrollings of the loop (visible in the output file `output/Counterexample.1.core.txt`):

$$\texttt{(n, z): (1, 2)} \rightarrow \underline{\texttt{(0, 0)}} \rightarrow \texttt{(-1, 1)} \rightarrow \texttt{(-2, 2)} \rightarrow \underline{\texttt{(0, 0)}}$$

The unrolling represents an execution with assignment `z = 2` at line 4. By inspecting the values of `n` and `z` at the loop-head location of each iteration, we see that the state `(n,z) = (0,0)` is visited twice. This represents a non-terminating loop.

Lasso-Based Analysis. The main idea of the lasso-based analysis is to extract potentially non-terminating structures called *lassos* and then pass each of them to the library LassoRanker [67]. This library constructs ranking functions, which are arguments for termination. Simultaneously, it is looking for a non-termination argument. If it finds a non-termination argument for at least one lasso, CPAchecker claims that the program is non-terminating.

The lasso-based analysis complements the termination-as-safety analysis. The analysis can verify that program `example-terminating.c` in Fig. 10a terminates, but not that program `example-nonterminating.c` in Fig. 10b might not terminate. The following command line runs the lasso-based analysis on the program in Fig. 10a (cf. example lassoRankerAnalysis on the web service):

```
cpachecker --lassoRankerAnalysis examples/example-terminating.c
```

CPAchecker reports the verdict `TRUE` and produces the output file `output/terminationAnalysisResult.txt`. This contains a termination argument in the form of the ranking function $3{*}z-3{*}n+4$. As n is always positive, if the loop condition $n \leq z$ is satisfied, $3*z-3*n+4 \geq 0$ holds. In addition, after each loop iteration, the resulting value of the ranking function strictly decreases. After a finite number of iterations, the value will eventually become smaller than zero,which implies the negation of the loop condition and thus termination.

4.10 Integer-Overflow Detection

To detect integer overflows, CPAchecker uses a standard reachability analysis, such as those explained in Sects. 4.2, 4.5, and 4.6, together with an internal encoding of overflow conditions as error locations (CPAchecker's overflow analysis also checks for underflows). The configurations supporting overflow detection have the suffix `--overflow` in their names. By default, CPAchecker only checks for signed integer overflows, as these are declared undefined behavior by the C standard. To additionally check for unsigned integer overflows, set the option `overflow.checkUnsigned` to `true`. For instance, to determine whether the example program in Fig. 2a is free of signed and unsigned integer overflows while using predicate abstraction (cf. Sect. 4.5), run the command (cf. example predicateAnalysis-unsigned-overflow on the web service):

```
cpachecker --predicateAnalysis--overflow \
    --option overflow.checkUnsigned=true examples/example-safe.c
```

The verification verdict is `FALSE`, because an overflow could happen at line 6 if `n` and `x` are initialized to 0 and 1, respectively.

5 Conclusion

This tutorial gives an introduction to the CPAchecker framework and how to use it to verify programs. It gives an overview of the main analysis techniques that CPAchecker offers, together with their strengths and weaknesses, and provides guidance on how to use CPAchecker in several analysis situations.

We hope that our tutorial is useful for researchers, practitioners, and educators, and that we stimulate interest and curiosity to dig deeper into the full potential of software model checking. Interested readers can find more information on the CPAchecker project web page, in the research publications on CPAchecker, the CPAchecker GitLab repository, and the CPAchecker mailing list.

Data-Availability Statement. CPAchecker is available at its project website https://cpachecker.sosy-lab.org and Zenodo [49]. This tutorial uses version 3.0 [50]. We also provide a reproduction package [6] that includes all the examples from this tutorial.

Funding Statement. CPAchecker was funded in part by the Canadian Natural Sciences and Engineering Research Council (NSERC) — 482301, by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY), 418257054 (Coop), 496588242 (IdeFix), 496852682 (ReVeriX), by the Free State of Bavaria, and by the LMU PostDoc Support Funds.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
2. Andrianov, P., Friedberger, K., Mandrykin, M.U., Mutilin, V.S., Volkov, A.: CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions (competition contribution). In: Proc. TACAS. pp. 355–359. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_22
3. Andrianov, P., Mutilin, V., Khoroshilov, A.: CPALockator: Thread-modular approach with projections (competition contribution). In: Proc. TACAS (2). pp. 423–427. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_25
4. Apel, S., Beyer, D., Mordan, V.O., Mutilin, V.S., Stahlbauer, A.: On-the-fly decomposition of specifications in software model checking. In: Proc. FSE. pp. 349–361. ACM (2016). https://doi.org/10.1145/2950290.2950349
5. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. Springer (2024)
6. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Reproduction package for FM 2024 article 'Software verification with CPAchecker 3.0: Tutorial and user guide'. Zenodo (2024). https://doi.org/10.5281/zenodo.13612338
7. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software verification with CPAchecker 3.0: Tutorial and user guide (extended version). arXiv/CoRR **2409**(02094) (September 2024). https://doi.org/10.48550/arXiv.2409.02094
8. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPAchecker 2.3 with strategy selection (competition contribution). In: Proc. TACAS (3). pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
9. Ball, T., Rajamani, S.K.: SLIC: A specification language for interface checking (of C). Tech. Rep. MSR-TR-2001-21, Microsoft Research (2002). https://www.microsoft.com/en-us/research/publication/slic-a-specification-language-for-interface-checking-of-c/
10. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., University of Iowa (2010). https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf

11. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
12. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
13. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15
14. Beyer, D., Chien, P.C., Jankola, M., Lee, N.Z.: A transferability study of interpolation-based hardware model checking for software verification. Proc. ACM Softw. Eng. **1**(FSE) (2024). https://doi.org/10.1145/3660797
15. Beyer, D., Chien, P.C., Lee, N.Z.: CPA-DF: A tool for configurable interval analysis to boost program verification. In: Proc. ASE. pp. 2050–2053. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00213
16. Beyer, D., Chien, P.C., Lee, N.Z.: Augmenting interpolation-based model checking with auxiliary invariants. In: Proc. SPIN. Springer (2024)
17. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The Blast query language for software verification. In: Proc. SAS. pp. 2–18. LNCS 3148, Springer (2004). https://doi.org/10.1007/978-3-540-27864-1_2
18. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proc. FMCAD. pp. 25–32. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351147
19. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579
20. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
21. Beyer, D., Dangl, M., Wendler, P.: Combining k-induction with continuously-refined invariants. Tech. Rep. MIP-1503, University of Passau (January 2015). https://doi.org/10.48550/arXiv.1502.00096
22. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
23. Beyer, D., Friedberger, K.: Domain-independent multi-threaded software model checking. In: Proc. ASE. pp. 634–644. ACM (2018). https://doi.org/10.1145/3238147.3238195
24. Beyer, D., Friedberger, K.: In-place vs. copy-on-write CEGAR refinement for block summarization with caching. In: Proc. ISoLA. pp. 197–215. LNCS 11245, Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_14
25. Beyer, D., Friedberger, K.: Domain-independent interprocedural program analysis using block-abstraction memoization. In: Proc. ESEC/FSE. pp. 50–62. ACM (2020). https://doi.org/10.1145/3368089.3409718
26. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
27. Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing software verification into off-the-shelf components: An application to CEGAR. In: Proc. ICSE. pp. 536–548. ACM (2022). https://doi.org/10.1145/3510003.3510064

28. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV. pp. 504–518. LNCS 4590, Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
29. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE. pp. 29–38. IEEE (2008). https://doi.org/10.1109/ASE.2008.13
30. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
31. Beyer, D., Jakobs, M.C.: Fred: Conditional model checking via reducers and folders. In: Proc. SEFM. pp. 113–132. LNCS 12310, Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_7
32. Beyer, D., Jakobs, M.C.: Cooperative verifier-based testing with CoVeriTest. Int. J. Softw. Tools Technol. Transfer **23**(3), 313–333 (2021). https://doi.org/10.1007/s10009-020-00587-8
33. Beyer, D., Jakobs, M.C., Lemberger, T.: Difference verification with conditions. In: Proc. SEFM. pp. 133–154. LNCS 12310, Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_8
34. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
35. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
36. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010). https://dl.acm.org/doi/10.5555/1998496.1998532
37. Beyer, D., Kettl, M., Lemberger, T.: Decomposing software verification using distributed summary synthesis. Proc. ACM Softw. Eng. **1**(FSE) (2024). https://doi.org/10.1145/3660766
38. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. J. Autom. Reasoning (2024). https://doi.org/10.1007/s10817-024-09702-9, preprint: https://doi.org/10.48550/arXiv.2208.05046
39. Beyer, D., Lemberger, T.: Symbolic execution with CEGAR. In: Proc. ISoLA. pp. 195–211. LNCS 9952, Springer (2016). https://doi.org/10.1007/978-3-319-47166-2_14
40. Beyer, D., Lemberger, T.: CPA-SymExec: Efficient symbolic execution in CPAchecker. In: Proc. ASE. pp. 900–903. ACM (2018). https://doi.org/10.1145/3238147.3240478
41. Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M.: A unifying approach for control-flow-based loop abstraction. In: Proc. SEFM. pp. 3–19. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_1
42. Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M.: CEGAR-PT: A tool for abstraction by program transformation. In: Proc. ASE. pp. 2078–2081. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00215
43. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11

44. Beyer, D., Löwe, S., Wendler, P.: Refinement selection. In: Proc. SPIN. pp. 20–38. LNCS 9232, Springer (2015). https://doi.org/10.1007/978-3-319-23404-5_3
45. Beyer, D., Petrenko, A.K.: Linux driver verification. In: Proc. ISoLA. pp. 1–6. LNCS 7610, Springer (2012). https://doi.org/10.1007/978-3-642-34032-1_1
46. Beyer, D., Stahlbauer, A.: BDD-based software model checking with CPAchecker. In: Proc. MEMICS. pp. 1–11. LNCS 7721, Springer (2013). https://doi.org/10.1007/978-3-642-36046-6_1
47. Beyer, D., Stahlbauer, A.: BDD-based software verification: Applications to event-condition-action systems. Int. J. Softw. Tools Technol. Transfer **16**(5), 507–518 (2014). https://doi.org/10.1007/s10009-014-0334-1
48. Beyer, D., Wendler, P.: CPAchecker with sequential combination and strategy selection. In: Automatic Software Verification. Springer (2024)
49. Beyer, D., Wendler, P.: CPAchecker releases. Zenodo. https://doi.org/10.5281/zenodo.3816620
50. Beyer, D., Wendler, P.: CPAchecker release 3.0. Zenodo (2024). https://doi.org/10.5281/zenodo.12663059
51. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
52. Bürdek, J., Lochau, M., Bauregger, S., Holzer, A., von Rhein, A., Apel, S., Beyer, D.: Facilitating reuse in multi-goal test-suite generation for software product lines. In: Proc. FASE. pp. 84–99. LNCS 9033, Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_6
53. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643
54. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: Proc. POPL. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
55. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS. pp. 423–425. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_34
56. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Proc. SAS. pp. 215–237. LNCS 7935, Springer (2013). https://doi.org/10.1007/978-3-642-38856-9_13
57. Friedberger, K.: CPA-BAM: Block-abstraction memoization with value analysis and predicate analysis (competition contribution). In: Proc. TACAS. pp. 912–915. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_58
58. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Proc. ATVA. pp. 365–380. LNCS 8172, Springer (2013). https://doi.org/10.1007/978-3-319-02444-8_26
59. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021
60. Jakobs, M.C.: CoVeriTest with dynamic partitioning of the iteration time limit (competition contribution). In: Proc. FASE. pp. 540–544. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_30
61. Jakobs, M.C.: CoVeriTest: Interleaving value and predicate analysis for test-case generation (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 847–851 (December 2021). https://doi.org/10.1007/s10009-020-00572-1

62. Jakobs, M.C., Richter, C.: CoVeriTest with adaptive time scheduling (competition contribution). In: Proc. FASE. pp. 358–362. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_18
63. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Handbook of Model Checking, pp. 447–491. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_15
64. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
65. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
66. Leeson, W., Dwyer, M.: Graves-CPA: A graph-attention verifier selector (competition contribution). In: Proc. TACAS (2). pp. 440–445. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_28
67. Leike, J., Heizmann, M.: Ranking templates for linear loops. Logical Methods in Computer Science **11**(1) (2015). https://doi.org/10.2168/LMCS-11(1:16)2015
68. Löwe, S.: CPAchecker with explicit-value analysis based on CEGAR and interpolation (competition contribution). In: Proc. TACAS. pp. 610–612. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_44
69. Löwe, S., Mandrykin, M.U., Wendler, P.: CPAchecker with sequential combination of explicit-value analyses and predicate analyses (competition contribution). In: Proc. TACAS. pp. 392–394. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_27
70. Löwe, S., Wendler, P.: CPAchecker with adjustable predicate analysis (competition contribution). In: Proc. TACAS. pp. 528–530. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_40
71. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV. pp. 1–13. LNCS 2725, Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
72. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV. pp. 123–136. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_14
73. Peled, D.: Ten years of partial order reduction. In: Proc. CAV. pp. 17–28. Springer (1998). https://doi.org/10.1007/BFb0028727
74. Richter, C., Wehrheim, H.: PeSCo: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS (3). pp. 229–233. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19
75. Ruland, S., Lochau, M., Jakobs, M.C.: HybridTiger: Hybrid model checking and domination-based partitioning for efficient multi-goal test-suite generation (competition contribution). In: Proc. FASE. pp. 520–524. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_26
76. Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. Electr. Notes Theor. Comput. Sci. **149**(1), 79–96 (2006). https://doi.org/10.1016/j.entcs.2005.11.018
77. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proc. FMCAD, pp. 127–144. LNCS 1954, Springer (2000). https://doi.org/10.1007/3-540-40922-X_8
78. The Open Group: 64-bit and data size neutrality. https://unix.org/whitepapers/64bit.html, accessed: 2024-06-29
79. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Proc. FMCAD. pp. 1–8. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351148

80. Vizel, Y., Grumberg, O., Shoham, S.: Intertwined forward-backward reachability analysis using interpolants. In: Proc. TACAS. pp. 308–323. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_22
81. Wendler, P.: CPAchecker with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In: Proc. TACAS. pp. 613–615. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_45
82. Wonisch, D.: Block abstraction memoization for CPAchecker (competition contribution). In: Proc. TACAS. pp. 531–533. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_41
83. Wonisch, D., Wehrheim, H.: Predicate analysis with block-abstraction memoization. In: Proc. ICFEM. pp. 332–347. LNCS 7635, Springer (2012). https://doi.org/10.1007/978-3-642-34281-3_24
84. Zakharov, I.S., Mandrykin, M.U., Mutilin, V.S., Novikov, E., Petrenko, A.K., Khoroshilov, A.V.: Configurable toolset for static verification of operating systems kernel modules. Programming and Comp. Softw. **41**(1), 49–64 (2015). https://doi.org/10.1134/S0361768815010065

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Satisfiability Modulo Theories: A Beginner's Tutorial

Clark Barrett[1](✉), Cesare Tinelli[2], Haniel Barbosa[3], Aina Niemetz[1], Mathias Preiner[1], Andrew Reynolds[2], and Yoni Zohar[4]

[1] Stanford University, Stanford, USA
barrett@cs.stanford.edu
[2] The University of Iowa, Iowa City, USA
[3] Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
[4] Bar-Ilan University, Ramat Gan, Israel

Abstract. Great minds have long dreamed of creating machines that can function as general-purpose problem solvers. Satisfiability modulo theories (SMT) has emerged as one pragmatic realization of this dream, providing significant expressive power and automation. This tutorial is a beginner's guide to SMT. It includes an overview of SMT and its formal foundations, a catalog of the main theories used in SMT solvers, and illustrations of how to obtain models and proofs. Throughout the tutorial, examples and exercises are provided as hands-on activities for the reader. They can be run using either Python or the SMT-LIB language, using either the cvc5 or the Z3 SMT solver.

1 Introduction

Great minds have long dreamed of creating machines that can reason deductively, that is, from a set of assumptions, determine whether a particular conclusion logically follows. The question of whether such a machine is possible was posed formally as a grand challenge by the famous mathematician David Hilbert in 1928, who called it the "Entscheidungsproblem" (decision problem) [24]. In 1936, both Church and Turing showed that, in general, this is impossible—the problem is undecidable [13,42]. Undeterred, researchers in automated reasoning have searched for ways to solve either special cases of the problem that are decidable or to find heuristics that work well in practice. Satisfiability modulo theories (SMT) has emerged as an approach that seems to fill a sweet spot in this search space. SMT leverages a rich collection of decidable theories to provide considerable expressive power without sacrificing decidability. SMT also permits some queries over problems that are undecidable or whose decidability is unknown. For these, it employs powerful heuristics that often work well in practice.

This tutorial is an introduction to SMT for new users. We explain what kinds of problems are suitable for SMT solvers, describe the capabilities of modern solvers, and provide guidance on how to encode problems as SMT queries.

© The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 571–596, 2025.
https://doi.org/10.1007/978-3-031-71177-0_31

Throughout the tutorial, we provide examples and exercises to illustrate the concepts being explained. Unless otherwise stated, the exercises can be completed using either the cvc5 [3] or the Z3 SMT solver [32], through either their Python interface or their textual interface based on the SMT-LIB 2 format [8]. The cvc5 website at cvc5.github.io contains documentation that can be used as a reference to supplement the material in this tutorial. An online version of the tutorial is also available on that site by clicking on Tutorials. To work through the examples and exercises, we recommend one of the following options.

A) To use a Python API for SMT, first create a virtual environment.

```
python3 -m venv smt-tutorial
source smt-tutorial/bin/activate
```

Next, install cvc5's Python API or Z3's Python API, or both.

```
python3 -m pip install cvc5
python3 -m pip install z3-solver
```

cvc5 is distributed under the BSD 3-clause license. Some features, however, such as its finite field solver (see Sect. 4.9), are only available in an extended version of cvc5 distributed under the GNU General Public License (GPL).[1] Since GPL is a problem for some users, the GPL version is not built or distributed by default. To install the GPL version of cvc5, use:

```
python3 -m pip install cvc5-gpl
```

Once a solver API is installed, you can copy example Python code into a script file, e.g., `Example.py`, and then type:

```
python3 Example.py
```

Note that, for the examples below, if you are using Z3 instead of cvc5, you must replace the first line of each Python code snippet with:

```
from z3 import *
```

B) Executables for cvc5 and Z3 are available for download. For cvc5, go to the cvc5 website, click on Downloads, and follow the link to the release page on GitHub. Alternatively, for Z3, go to the Z3 releases page at github.com/Z3Prover/z3/releases. From either release page, download the latest release compatible with your machine (for cvc5, choose a GPL download if you want support for finite fields). Once you unzip the downloaded archive, the executable will be in the `bin` directory. Thus, if the unzipped directory is called `release-dir`, and you have downloaded cvc5, you can run an SMT-LIB example called `Example.smt2` by typing:

```
release-dir/bin/cvc5 Example.smt2
```

from your shell's command line. If you downloaded Z3, type instead:

[1] The finite field solver uses the CoCoA library [1], which has a GPL license.

```
release-dir/bin/z3 Example.smt2
```

C) From the cvc5 website, click on Try cvc5 online. This links to a page that provides a web interface for running cvc5 on scripts in the SMT-LIB format.

This tutorial has been tested with cvc5 1.2.0, Z3 4.13.0, and Python 3.12.3, but later releases should work as well. Solver outputs shown below are based on cvc5 version 1.2.0. Other versions or solvers should produce conceptually similar results, but the outputs may not be exactly the same. The SMT-LIB examples are based on version 2.6 of the format [5]. cvc5's Python API was designed to be a drop-in replacement for Z3's Python API. The credit for the design of the Python API thus goes to the Z3 authors.

2 Overview

At an intuitive level, SMT solvers are general-purpose problem solving tools. They are somewhat similar to calculators, in that the user provides the problem of interest, and the tool does some calculation to produce an answer. However, they are much more powerful than a simple calculator.

SMT solvers reason *symbolically*, as is done in grade school algebra. The user provides a set of *assertions* that describe constraints to be satisfied, and the solver produces a *solution* satisfying *all* of the constraints, if there is one. Consider the following simple example, mimicking a typical algebra word problem.

Example 1. In 10 years, Alice will be twice as old as Bob is now, but in 22 years, Bob will be twice as old as Alice is now. How old are Alice and Bob?

First, let's see how to solve this using Python.

```
from cvc5.pythonic import *
a, b = Ints('a b')
solve(a + 10 == 2 * b, b + 22 == 2 * a)
```

The Pythonic API is designed to be as simple and intuitive as possible. We introduce the symbols we are using (SMT solvers always require that symbols be introduced before they are used), and then we call `solve`, passing in the two equations in much the same way we would write them naturally. The output is a simple representation of the solution as a Python list.

```
[a = 18, b = 14]
```

Alternatively, SMT solvers can take as input a script written in the SMT-LIB language [5], a standard developed by the SMT community whose syntax is similar to that of LISP. Below is the same example written in SMT-LIB.

```
(set-logic QF_LIA)
(set-option :produce-models true)

(declare-const a Int)
(declare-const b Int)
```

```
(assert (= (+ a 10) (* 2 b)))
(assert (= (+ b 22) (* 2 a)))

(check-sat)
(get-model)
```

The result is:

```
sat
(
(define-fun a () Int 18)
(define-fun b () Int 14)
)
```

Notice that the solver replies `sat` before giving the solution. This is short for "satisfiable," a word meaning that there is at least one solution. SMT solvers can also identify when a set of assertions has no solution. In this case, the solver replies `unsat`, which is short for "unsatisfiable."

Let's take a closer look at the SMT-LIB input file, which is a sequence of *commands*. The command in the first line tells the solver which *logic* we are working in. In this case, we are using `QF_LIA` which stands for quantifier-free linear integer arithmetic. We explain more about logics in Sect. 4 below. The second line tells the solver to produce *models*. A model assigns a concrete meaning to every user-declared symbol. Without turning this option on, a solver will still respond with `sat` or `unsat`, but it may not be able to provide a model. The next two lines declare two *uninterpreted constants* called `a` and `b`. Informally, we often refer to these as variables, because they play the same role that variables do in math. However, in the automated reasoning literature, a *variable* typically refers to a symbol that is bound by a quantifier, whereas an uninterpreted constant is a symbol whose value is determined by a model. SMT-LIB follows the the latter terminology. The next two lines create *assertions*. An assertion is a way of telling the solver about a formula that we would like to be true in the model that is produced. Note that the formulas too are specified in a LISP-like prefix syntax. Finally, the command `(check-sat)` tells the solver to check whether the set of assertions made so far is satisfiable, and the command `(get-model)` (which is only legal if the solver returns `sat`) prints values for each uninterpreted constant, with the guarantee that assigning these values to the constants makes all the assertions true. The values are printed using legal SMT-LIB syntax in case the user wants to copy and paste them into a new SMT-LIB script.

Exercise 1. Consider a modification of Example 1. The first assertion will stay the same, but for the second, let's assert that Bob will be twice as old as Alice in only 20 years. Modify the Python program or SMT-LIB script to reflect the new set of constraints. What output does the SMT solver give?

So far, we have seen the most basic use of an SMT solver. Given a set of assertions, determine whether there is a solution for them. We now show that this basic capability can be used to answer several similar questions.

Suppose we have a set X of assumptions about the world, and we want to know whether some hypothetical Y is possible under those assumptions. If we can express X and Y as SMT formulas, then an SMT solver can answer the question. In fact, we simply assert each assumption in X as well as the formula representing Y and check whether this set of assertions is satisfiable.

Example 2. Let x and y be 32-bit integers, with x a multiple of 2. Is it possible for the machine arithmetic product of x and y to be 1?

For this problem, we'll use *bit-vectors*. SMT solvers use bit-vectors to model machine arithmetic and other operations on fixed-size vectors of bits. The SMT-LIB encoding is as follows.

```
(set-logic QF_BV)

(declare-const x (_ BitVec 32))
(declare-const y (_ BitVec 32))
(declare-const z (_ BitVec 32))

(assert (= x (bvmul z (_ bv2 32))))
(assert (= (bvmul x y) (_ bv1 32)))

(check-sat)
```

This time, we use the logic `QF_BV` which stands for quantifier-free bit-vectors. The underscore symbol `_` is used in SMT-LIB to indicate that the next symbol is indexed by the following argument. It is used to specify the bit-vector size in this example. The `bvmul` symbol represents bit-vector multiplication, and the notation `bvX` is the bit-vector constant whose value, in decimal notation, is `X`. Constant `z` names the value we must multiply by 2 to get x. Here's how to solve it using the Pythonic API. This time, we'll use the API in a way that more closely resembles the SMT-LIB script.

```
from cvc5.pythonic import *

x, y, z = BitVecs('x y z', 32)
s = SolverFor('QF_BV')

s.add(x == z * 2)
s.add(x * y == 1)

result = s.check()
print("result: ", result)
```

There is no solution because an even number does not have a multiplicative inverse in machine arithmetic (i.e., when doing arithmetic modulo a power of 2).

Exercise 2. Find the multiplicative inverse of 5 (mod 2^8).

Another common situation is when we have a set X of assumptions, and we want to know whether some Y *must* hold as a consequence. If so, we say that Y is *implied* or *entailed* by X. Again, assuming we can represent X and Y using formulas, we can start by asserting the formulas representing X. At this point, however, we do not assert the formula for Y. Instead, we assert its *negation*. If the result is `unsat`, then Y must follow from X. The reasoning is that if it is not possible for the negation of Y to be true when X is true, then Y itself must be true. Let's look at a version of the well-known syllogism about Socrates.

Example 3. If all humans are mortal, and Socrates is a human, then must Socrates be mortal?

The Python code is as follows.

```
from cvc5.pythonic import *
S = DeclareSort("S")
Bool = BoolSort()
Human = Function("Human", S, Bool)
Mortal = Function("Mortal", S, Bool)
Socrates = Const("Socrates", S)

s = SolverFor('UF')

x = Const("x", S)
s.add(ForAll([x], Implies(Human(x), Mortal(x))))
s.add(Human(Socrates))
s.add(Not(Mortal(Socrates)))

print(s.check())
```

The SMT-LIB version of the same problem looks like this.

```
(set-logic UF)

(declare-sort S 0)
(declare-fun Human (S) Bool)
(declare-fun Mortal (S) Bool)
(declare-const Socrates S)

(assert (forall ((x S)) (=> (Human x) (Mortal x))))
(assert (Human Socrates))
(assert (not (Mortal Socrates)))

(check-sat)
```

This problem illustrates a few new encoding tools. First, we use the logic `UF` which stands for "uninterpreted functions." This logic allows us to declare new function symbols. Note that it is also missing the `QF` prefix we've used above, which means that quantifiers are also allowed. We declare a new uninterpreted *sort* `S`. A sort is like a type in programming languages. We use an *uninterpreted* sort to represent a class of individual objects that cannot be modeled with the predefined sorts provided by SMT-LIB, (so far, we've seen the predefined sorts for integers and bit-vectors). Next, we declare two functions, `Human` and `Mortal`, each of which takes a single argument of sort `S` and returns a `Bool`, the SMT-LIB Boolean sort. A function returning a Boolean is also called a *predicate*. We then declare an uninterpreted constant called `Socrates` of sort `S`. Now, we are ready to encode the first fact, namely that all humans are mortal. To do so, we use the *universal quantifier*, `ForAll`. The assertion states that for every individual `x` of sort `S`, if the predicate `Human` holds for that individual, then the predicate `Mortal` also holds. The next assertion states that the `Human` predicate holds for `Socrates`. Finally, we want to see whether the fact that Socrates is mortal necessarily follows from the assumptions. To do this, we assert the negation of the statement and check for satisfiability. Running the example confirms that the result is unsatisfiable and thus, indeed, this statement is entailed.

What we have presented so far should provide a good high-level idea of what is possible with SMT solvers.[2] We cover these ideas in more detail in the following. In Sect. 3, we briefly describe the formal foundations for SMT. Next, in Sect. 4, we catalog the different *theories* supported by SMT solvers and provide examples of how to use them. We cover the different outputs produced by SMT solvers, including models and proofs, in Sect. 5, and conclude in Sect. 6 with pointers to additional resources.

3 Formal Foundations

The satisfiability modulo theories problem can be formalized in many-sorted first-order logic with equality. We briefly outline the necessary concepts here. Due to space constraints, we assume some familiarity with basic concepts and notation from mathematical logic. More details can be found in [21,25].

3.1 Syntax

In first-order logic, one constructs formulas that are statements about individuals in some domain of discourse and their relationships. Many-sorted logic adds the possibility of talking about multiple, separate domains.

Signatures. The language of formulas is determined by a vocabulary of symbols, called a *signature*, which has three main components: *sort symbols* (such as Int, Real, Person, etc.) which name, or *denote*, domains of interest; *function symbols* (such as, $+$, $*$, log, mother, father) which denote total functions over the domains; and *relation symbols* (such as, $=$, $<$, even, married) which denote total relations over the domains. A signature also specifies the *arity* of each function symbol f, which is the number of inputs f takes, as well as its *rank*, which consists of the sort of f's inputs and of f's output.[3] We say that f has arity n and rank $\sigma_1 \cdots \sigma_n \sigma$ in a signature Σ if f takes n inputs of respective sorts $\sigma_1, \ldots, \sigma_n$ and returns an output of sort σ. A function symbol of arity 0 and rank σ (such as 0, 1, true, ...) is also called a *constant symbol* of sort σ. It is convenient to consider only signatures that have a distinguished sort Bool, for the Booleans, and treat relation symbols as function symbols whose return type is Bool. In addition, we assume that every signature contains a distinguished function symbol $\approx_\sigma$ of rank $\sigma\sigma\mathsf{Bool}$, denoting the identity relation, for each sort σ of Σ.

A signature Σ is a *subsignature* of a signature Ω, and Ω is a *supersignature* of Σ, if all the sort and function symbols of Σ are also in Ω and the function symbols have the same rank in Ω as they do in Σ.

[2] More sophisticated features and use cases are beyond the scope of this tutorial, but we plan to provide additional tutorials on more advanced topics in the future.

[3] For simplicity, we do not consider the more general case where function symbols can be overloaded by being assigned more than one arity and/or rank.

Variables, Terms and Formulas. To build formulas, in addition to fixing a signature Σ, we also fix a set $\mathbf{X}$ of *sorted* variables, each associated with a sort σ and standing for some element from (the set denoted by) σ. We can then build terms out of variables and function symbols from Σ. Given a signature Σ, a *well-sorted Σ-term*, or just *term* for short, is defined inductively as follows: (i) a variable or constant symbol of sort σ is a term of sort σ; (ii) if f is a function symbol of rank $\sigma_1 \cdots \sigma_n \sigma$, with $n > 0$, and $t_1, \ldots, t_n$ are terms of sort $\sigma_1 \cdots \sigma_n$, respectively, then the expression $f(t_1, \ldots, t_n)$ is a term of sort σ; (iii) if φ is a term of sort Bool and x is a variable of sort σ, then the expressions $\exists\, x{:}\sigma.\, \varphi$ and $\forall\, x{:}\sigma.\, \varphi$ are terms of sort Bool. We then *identify formulas with terms of sort* Bool. The distinguished symbols $\forall$ and $\exists$ are *quantifier symbols*. We say that a variable x *occurs free* in a formula φ if x occurs in φ and either φ contains no quantifier symbols or it has the form $\exists\, y{:}\sigma.\, \varphi'$ or $\forall\, y{:}\sigma.\, \varphi'$, for some variable y, where x occurs free in φ'.

3.2 Semantics

For each signature Σ, the meaning of Σ-terms is provided by mathematical structures called interpretations. A *Σ-interpretation* $\mathcal{I}$ maps:

- each sort σ of Σ to a *non-empty* set $\sigma^{\mathcal{I}}$, the *domain* of σ in $\mathcal{I}$, with $\mathsf{Bool}^{\mathcal{I}}$ being the binary set $\{\mathsf{true}, \mathsf{false}\}$;
- each variable $x \in \mathbf{X}$ of sort σ to an element $x^{\mathcal{I}} \in \sigma^{\mathcal{I}}$;
- each function symbol f of rank $\sigma_1 \cdots \sigma_n \sigma$ to a *total* function $f^{\mathcal{I}}$ of type $\sigma_1^{\mathcal{I}} \times \cdots \times \sigma_n^{\mathcal{I}} \to \sigma^{\mathcal{I}}$ (and, in particular, each constant symbol c of sort σ to an element $c^{\mathcal{I}} \in \sigma^{\mathcal{I}}$).

We say that σ (resp. x, f) *denotes* the set $\sigma^{\mathcal{I}}$ (element $x^{\mathcal{I}}$, function $f^{\mathcal{I}}$) in $\mathcal{I}$. Every Σ-interpretation $\mathcal{I}$ extends from variables and function symbols to Σ-terms t as follows: (i) a term $f(t_1, \ldots, t_n)$ *evaluates* in $\mathcal{I}$ to $f^{\mathcal{I}}(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}})$, the value returned by function $f^{\mathcal{I}}$ when applied to the elements denoted by $t_1, \ldots, t_n$; (ii) an *existentially quantified* formula $\exists\, x{:}\sigma.\, \varphi$ evaluates to true in $\mathcal{I}$ if and only if φ evaluates to true in an interpretation $\mathcal{I}[x \mapsto a]$ that maps x to *some* suitable $a \in \sigma^{\mathcal{I}}$ and is otherwise identical to $\mathcal{I}$; (iii) a *universally quantified* formula $\forall\, x{:}\sigma.\, \varphi$ evaluates to true in $\mathcal{I}$ if and only if φ evaluates to true in $\mathcal{I}[x \mapsto a]$ for *all* possible choices of values for x in $\sigma^{\mathcal{I}}$.

An interpretation $\mathcal{I}$ *satisfies* a formula φ if $\varphi^{\mathcal{I}} = \mathsf{true}$ and *falsifies* it if $\varphi^{\mathcal{I}} = \mathsf{false}$. In the former case, we also say that $\mathcal{I}$ is a *model* of φ.

The *reduct* of an Ω-interpretation $\mathcal{I}$ to a subsignature Σ of Ω is the (unique) Σ-interpretation that interprets the symbols of Σ exactly as $\mathcal{I}$. Intuitively, the reduct is obtained by *forgetting* the symbols of Ω that are not in Σ.

In the definition of interpretation above, we have not provided a meaning for the usual Boolean connectives such as $\neg, \wedge, \vee, \Rightarrow$ and so on. In SMT, specific interpretations of function symbols are provided by a theory, as explained next.

3.3 Theories

In general, we are not interested in arbitrary interpretations of terms and formulas in a signature Σ but in interpretations belonging to a specific *theory* T that *constrain* the meaning of the symbols in Σ; for instance, that interpret $\neg$ and $\wedge$ as logical negation and conjunction, $0, 1, 2, \ldots$ as the natural numbers, and so on. Traditionally in logic, a theory is defined by a set of formulas, called *axioms*: one considers only Σ-interpretations that satisfy all the axioms. In SMT, a theory is, more generally, a class of interpretations that can be specified axiomatically or in other ways. More precisely, a *Σ-theory* T is a pair $(\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations, however specified. We describe and discuss several examples of theories commonly used in SMT in the next section.

Given a theory $T = (\Sigma, \mathbf{I})$, we consider not just Σ-formulas but Ω-formulas for some supersignature Ω of Σ. In the context of T, we refer to the symbols of Σ as *theory* symbols and to the additional symbols in Ω as *uninterpreted* symbols. For instance, in the theory of reals, we may write a formula of the form $a + 1 > b$ where a and b are uninterpreted, or *symbolic*, constants of sort Real. Intuitively, while the meaning of $+$ and 1 is fixed by the theory, the meaning of a and b is not. Hence, we consider the formula satisfiable if there are real values for a and b which make the formula evaluate to true. This idea is formalized in the notion of *satisfiability in T*.

Satisfiability Modulo a Theory. If T is a Σ-theory, a *T-interpretation* is *any* Ω-interpretation $\mathcal{I}$ for some supersignature Ω of Σ whose restriction to Σ differs from an interpretation of T at most in the way it interprets the variables.

An Ω-formula φ is *satisfiable in* T if it is satisfied by *some* T-interpretation $\mathcal{I}$—which may interpret the variables of φ and the sort, function, and predicate symbols not in Σ arbitrarily. The formula is *valid in T* if it is satisfied by *all* T-interpretations. A set Φ of Ω-formulas *entails φ in T*, written $\Phi \models_T \varphi$, if every T-interpretation that satisfies all formulas in Φ satisfies φ as well. The set Φ is *satisfiable in* T if there is a T-interpretation that satisfies all of its formulas.

4 SMT Theories

A key feature of SMT is that the entire problem is parameterized by the choice of a theory T. This is important because it means that SMT is an algorithmic *framework*, rather than a fixed algorithm. Thus, if a particular problem cannot easily be encoded in any existing theory supported by SMT solvers, one option is to add support for a new theory which is better suited to the problem. In fact, this is exactly the process by which many of the theories supported by modern SMT solvers were added.

Theories can be used alone or in arbitrary combinations. Besides the theory, other parameters related to the syntax of formulas include whether or not to enable quantifiers and whether to disallow or limit the use of certain theory operations. In the SMT-LIB standard, and in solvers that support it, these parameters are configured by specifying a *logic*. A logic identifies the theory (or theories) being used and optionally imposes syntactic restrictions on the allowed formulas. Users can provide the SMT solver with a predefined *logic name* (like `QF_LIA`, `QF_BV`, and `UF` seen earlier) to specify which logic is to be used. By default (i.e., if no logic name is provided), SMT solvers typically enable all the theories they support and allow all operations. This is equivalent to using the special logic name `ALL`. However, solvers are often tuned with specific heuristics for specific logics. Thus, it is advisable to provide the solver with the most specific logic name possible. In this section, we discuss the most common theories and logics supported by SMT solvers, with examples of each.

4.1 Core Theory and Uninterpreted Symbols

The SMT-LIB standard defines a *core theory* which consists of a core signature with a fixed interpretation that is always present, regardless of which other theories are being used. The core theory defines the Boolean sort `Bool` (`BoolSort()` in Python), the Boolean theory constants `true` and `false` (`BoolVal(True)` and `BoolVal(False)` in Python), and the operators `=`, `not`, `and`, `or`, `xor`, and `=>` (`==`, `Not`, `And`, `Or`, `Xor`, and `Implies` in Python), all with the usual meanings. The equality symbol `=` is *polymorphic*: it can be applied to two terms of the same sort, for any predefined or user-declared sort. There are also two more polymorphic operators that require a bit more explanation. The `distinct` (`Distinct`) operator takes two or more arguments of the same sort and returns `true` exactly when all the arguments have pairwise distinct values. The `ite` (`If`) operator takes three arguments, the first of which must be of Boolean sort. The other two arguments can have any sort as long as it is the same for both. The meaning of the `ite` operator is the second argument when the first argument is true, and the third argument otherwise.

The simplest logic that builds on the core theory is `QF_UF`, short for "quantifier-free uninterpreted functions." This logic disallows quantifiers and does not define any new symbols beyond those in the core theory. However, it allows the user to extend the signature with new sorts and symbols. The SMT solver is allowed to interpret these symbols in any way it chooses. This is why they are referred to as uninterpreted: the solver does not impose any restrictions on the interpretation (besides the declared arity and rank). The following example illustrates the use of uninterpreted symbols as well as the `And` and `Distinct` operators.

Example 4. Let f be a unary function from U to U, for some set U. Check that, whatever the meaning of f, if $f(f(f(x))) = x$ and $f(f(f(f(f(x))))) = x$, then $f(x) = x$.

We show a solution in Python followed by one using SMT-LIB.

```
from cvc5.pythonic import *
U = DeclareSort("U")
f = Function("f", U, U)
x = Const("x", U)
```

```
s = SolverFor('QF_UF')
s.add(And((f(f(f(x))) == x), (f(f(f(f(f(x))))) == x)))
s.add(Distinct(f(x), x))  # negation of f(x) = x
print(s.check())
```

```
(set-logic QF_UF)
(declare-sort U 0)

(declare-fun f (U) U)
(declare-const x U)

(assert (and (= (f (f (f x))) x) (= (f (f (f (f (f x))))) x)))
(assert (distinct (f x) x))

(check-sat)
```

We can derive $f(x) = x$ from the first assertion by performing a series of substitutions, and thus the problem is unsatisfiable. Now, we present a simple example that illustrates the `ite` operator. It also shows that in Python, we can use `!=` instead of `Distinct` to assert that two terms are distinct.

Example 5. Suppose we know that x is either equal to y or z, depending on the value of the Boolean b. Suppose we further know that w is equal to one of y or z. Does it follow that $x = w$?

The Python solution is shown below.[4]

```
from cvc5.pythonic import *
U = DeclareSort("U")
b = Const("b", BoolSort())
x, y, z, w = Consts("x y z w", U)

s = SolverFor('QF_UF')
s.add(x == (If(b, y, z)))
s.add(Or((w == y), (w == z)))
s.add(x != w)

if s.check() == sat:
  m = s.model()
  print("\n".join([str(k) + " : " + str(m[k]) for k in m]))
```

cvc5 outputs the following for this example.

```
b : True
w : (as @U_0 U)
x : (as @U_1 U)
y : (as @U_1 U)
z : (as @U_0 U)
```

The result tells us that it does *not* follow that $x = w$. The model gives us a *counterexample* to that claim. Because the sort `U` is uninterpreted, the model returned by cvc5 must choose an interpretation for it. Here, cvc5 tells us that it is interpreting `U` as a set with two elements, named `@U_0` and `@U_1`. The model

[4] Due to space constraints, the SMT-LIB versions of the remaining examples do not appear in the text. They are available in the online version of the tutorial available from the Tutorials link on the cvc5 website.

then specifies that x and y have one value and z and w have the other, so x is not equal to w.

Exercise 3. Modify Example 4 to make it satisfiable and Example 5 to make it unsatisfiable.

4.2 Arithmetic

Though there are many tools available for arithmetic reasoning, SMT solvers are unique in their ability to reason efficiently about arbitrary Boolean combinations of arithmetic constraints, as well as to combine arithmetic reasoning with reasoning about other theories. It is important to note that SMT solvers reason *precisely* about both integer and real arithmetic. That is, they use arbitrary-precision arithmetic as opposed to machine integer or floating-point approximations. This means that SMT solvers are not susceptible to the numerical errors that can arise, for instance, when using floating-point arithmetic to approximate real arithmetic. It also means that for problems whose complexity lies mainly in the arithmetic reasoning, as opposed to Boolean reasoning, SMT solvers are typically slower than tools that use floating-point approximations. The underlying algorithms for arithmetic reasoning in SMT solvers are based on standard techniques that have been adapted to the SMT context, such as the Simplex algorithm [20] and Cylindrical Algebraic Decomposition [2].

There are a large number of logics to choose from within the arithmetic umbrella, with reasoning over reals generally more efficient than reasoning over integers, and reasoning over less expressive formulas generally more efficient than reasoning over more expressive ones. We briefly discuss the various logics here.

Difference Logic. In *difference logic*, every arithmetic constraint must be of the form $x-y \bowtie c$ or $x \bowtie c$, where $\bowtie \in \{=, <, >, \leq, \geq\}$, and c is a numeric theory constant. If x and y range over integers, we call it *integer difference logic*, and if they range over reals, we call it *real difference logic*. Efficient algorithms exist for both [17,36]. The names of these logics are QF_IDL and QF_RDL, respectively. One application for difference logic is *job shop scheduling* [41].

Example 6. Suppose we have 3 jobs to complete on 2 machines. Job 1 requires machine 1 for 10 min and then machine 2 for 5 min. Job 2 requires machine 2 for 20 min and then machine 1 for 5 min. And Job 3 requires machine 1 for 5 min and then machine 2 for 5 min. Can all jobs be completed in 30 min?

To solve the problem, we create integer variables for the start times of each task within each job. We assert that the start times are non-negative, each task within each job doesn't start until the previous task finishes, and tasks on each machine don't overlap. Finally, we check that each task finishes on time.

```
from cvc5.pythonic import *

j11,j12,j21,j22,j31,j32 = Ints("j11 j12 j21 j22 j31 j32")
```

```
s = SolverFor('QF_IDL')
s.add(And([x >= 0 for x in [j11, j12, j21, j22, j31, j32]]))
s.add(And(j12 - j11 >= 10, j22 - j21 >= 20, j32 - j31 >= 5))
s.add(And(Or(j22 - j11 >= 10, j11 - j22 >= 5),
          Or(j31 - j11 >= 10, j11 - j31 >= 5),
          Or(j31 - j22 >= 5, j22 - j31 >= 5)))
s.add(And(Or(j21 - j12 >= 5, j12 - j21 >= 5),
          Or(j32 - j12 >= 5, j12 - j32 >= 5),
          Or(j32 - j21 >= 5, j21 - j32 >= 5)))
s.add(And(j12 <= 25, j22 <= 25, j32 <= 25))

print(s.model() if s.check() == sat else "unsat")
```

Exercise 4. What is the minimum amount of time that it will take to complete all of the jobs in Example 6?

Linear Arithmetic. The logic of linear arithmetic allows arithmetic constraints to have any form that is equivalent to $\sum c_i x_i + b \bowtie 0$, where b, c_i are numeric theory constants and $\bowtie \in \{=, <, >, \leq, \geq\}$. As before, there are both integer and real variants, `QF_LIA` and `QF_LRA`, respectively. One can also mix the two with `QF_LIRA`. Note that, according to the SMT-LIB standard, when using `QF_LIRA`, integers and reals should not be mixed in the same linear sum, but most solvers (including cvc5 and Z3) are more permissive and do allow mixed terms. Example 1 is a good example of a simple `QF_LIA` problem.

Exercise 5. Repeat Exercise 1, but change the logic to `QF_LRA`, change the types of the variables from `Int` to `Real`, and append `.0` to each numeric constant. Now, what output does the solver give?

Nonlinear Arithmetic. Moving up the expressiveness hierarchy, we next have logics for quantifier-free *nonlinear arithmetic*. In these logics, arbitrary polynomials are allowed in constraints. The logic `QF_NRA` is for nonlinear arithmetic over the reals, which is decidable but with doubly exponential complexity [2]. On the other hand, the same logic over integers, `QF_NIA`, is undecidable. cvc5 implements a decision procedure for `QF_NRA` based on a combination of heuristic pruning and cylindrical algebraic coverings [29]. cvc5 and other tools implement incomplete heuristic procedures for `QF_NIA`.

Example 7. Find a solution for $x^2y + yz + 2xyz + 4xy + 8xz + 16 = 0$.

```
from cvc5.pythonic import *
x, y, z = Reals("x y z")
s = SolverFor('QF_NRA')
s.add(x*x*y + y*z + 2*x*y*z + 4*x*y + 8*x*z + 16 == 0)
print(s.model() if s.check() == sat else "unsat")
```

4.3 Arrays

Consider the following Python function which swaps two elements in a dictionary.

```
def swap(a,i,j):
  tmp = a[i]
  a[i] = a[j]
  a[j] = tmp
```

If a[i] and a[j] happen to be equal, the dictionary a is unchanged by the function. To prove this fact, we could try modeling dictionaries as uninterpreted functions. However, asserting that two functions are equal is not allowed in first-order logic. Alternatively, we could use a quantifier to assert that two functions return the same output when given the same input, for any input. However, we would like to avoid quantifiers when possible, as their use puts us in an undecidable logic.

Fortunately, the SMT-LIB standard includes a theory of arrays [30], which can help in this situation. The theory is perhaps more accurately viewed as a theory of mutable maps and is parameterized by two sorts, one for the index (corresponding to the key type of the dictionary) and one for the elements (values in the dictionary). For example, the SMT-LIB sort (Array Int Real) represents arrays indexed by integers and containing reals. Note that SMT arrays are always total, in the sense that they have an element for every value in the index sort. In particular, an array indexed by Int is conceptually infinite.

The theory has two operators: select, which takes an array and an index and returns the element at that index, and store, which takes an array a, an index i, and an element e, and returns a new array that is the result of updating a with the element e at index i.

Typically, the theory of arrays is used in combination with other theories that make sense for the index and element sorts. For example, the logic QF_ALIA allows quantifier-free formulas with variables that range over integers and arrays of integers. The simplest logic with arrays is QF_AX, in which all the sorts must be uninterpreted.

In the example below, we encode the above problem using the array theory.

Example 8. For the Python program above, show that, for arbitrary index and element sorts, if a[i] and a[j] are equal, then so are a and swap(a,i,j).

```
from cvc5.pythonic import *
I = DeclareSort("I")
E = DeclareSort("E")
i, j = Consts("i j", I)
tmp = Const("tmp", E)
array = ArraySort(I, E)
a_in, a_out = Consts("a_in, a_out", array)

s = SolverFor('QF_AX')

s.add(tmp == (Select(a_in, i)))
s.add(a_out == (Store(Store(a_in, i, Select(a_in, j)),
    j, tmp)))
s.add((Select(a_in, i)) == (Select(a_in, j)))
s.add(a_in != a_out)

print(s.check())
```

Exercise 6. Another property of swap that we can prove is that if a[i] and a[j] are distinct, then swap would change a. Modify the solution for Example 8 to prove this property.

4.4 Bit-Vectors

Consider a simple implementation (written in a C-like syntax) for computing the absolute value of a 32-bit integer: $abs(x) \coloneqq x < 0 \; ? \; -x : x$. Instead of branching on $x < 0$, it is possible to compute the absolute value of x with three or four branch-free operations [28] as follows. Let xrs be an abbreviation for the arithmetic right shift ($\gg_s$) of x by 31 bits. Note that the result of this operation is either 0 or -1 (all bits set to 1), depending on the most significant bit (MSB) of x: if the MSB of x is 0, xrs is 0; otherwise, xrs is -1. Three branchless alternatives for computing the absolute value of x are as follows.

1. $abs_1(x) \coloneqq (x \oplus xrs) - xrs$
2. $abs_2(x) \coloneqq (x + xrs) \oplus xrs$
3. $abs_3(x) \coloneqq x - ((x \ll 1) \mathbin{\&} xrs)$

These branchless versions of $abs(x)$ make use of the 32-bit versions of the bit-wise operations exclusive or ($\oplus$), bit-wise and ($\&$), logical shift left ($\ll$), and arithmetic shift right ($\gg_s$).

We can use an SMT solver to prove whether the branchless versions are equivalent to the original implementation. Note that integers, as discussed in Sect. 4.2, are not a good fit, as it is difficult to model the bitwise operators using the arithmetic operators. However, the SMT-LIB standard includes a theory of fixed-size bit-vectors, which defines the bit-precise semantics of fixed-size machine integers. The name for the quantifier-free logic containing just this theory is QF_BV. Using this logic, we can easily check the equivalence of the absolute value computations.

Example 9. Show that the first branchless alternative abs_1 is equivalent to abs.

```
from cvc5.pythonic import *
x = Const("x", BitVecSort(32))
xrs = x >> 31
s = SolverFor('QF_BV')
s.add(If(x < 0, -x, x) != (x ^ xrs) - xrs)      # prove abs() == abs1()
print(s.model() if s.check() == sat else "unsat")
```

Exercise 7. Show that the second and third branchless alternatives abs_2 and abs_3 are equivalent to abs.

4.5 Datatypes

Built into the SMT-LIB language is a mechanism for defining *(algebraic) datatypes*. Datatypes are highly useful in applications for reasoning about data structures like records, lists, and trees [7]. The quantifier-free logic name is QF_DT.

Example 10. Model a binary tree containing integer data. Find trees x and y such that (i) the left subtree of x is the same as the right subtree of y and (ii) the data stored in x is greater than 100.

Note that we need both datatypes and integer arithmetic for this example. cvc5 supports the logic name `QF_DTLIA`, but Z3 does not. Fortunately, we can always use `ALL` for the logic if a more specific logic is not available.

```
from cvc5.pythonic import *
decl = Datatype("tree")
decl.declare("node", ("data", IntSort()), ("left", decl), ("right", decl))
decl.declare("nil")
Tree = decl.create()

x, y = Consts("x y", Tree)

s = SolverFor('ALL')
s.add(Tree.is_node(x))
s.add(Tree.is_node(y))
s.add(Tree.left(x) == Tree.right(y))
s.add(Tree.data(x) > 100)

print(s.model() if s.check() == sat else "unsat")
```

The output gives the values for x and y.

```
[ x = node(101, nil, node(0, nil, nil)),
  y = node(0, node(0, nil, node(0, nil, nil)), nil)]
```

Exercise 8. Show that a tree cannot be equal to its own left subtree.

4.6 Floating-Point Arithmetic

The most common representation of real numbers in hardware and software is the binary floating-point number representation system as defined by the IEEE Standard 754-2019 for Floating-Point Arithmetic [27]. Floating-point numbers are encoded as a triple of bit-vectors: the fractional part (the significand), the exponent (a power of 10 by which the significand is multiplied), and a sign bit. This representation is of limited range and precision, and thus, the domain of floating-point numbers is finite. It also includes special values for representing errors as not-a-number and for plus and minus infinity. In SMT-LIB, the IEEE-754 standard is formalized as the theory of floating-point arithmetic [11]. The quantifier-free logic name is `QF_FP`.

Example 11. The SMT-LIB standard supports a *fused multiplication and addition* operator `fp.fma`. Given three single precision floating-point numbers a, b, and c, show that the floating-point fused multiplication and addition of a, b, and c is different from first multiplying a and b and then adding c.

```
from cvc5.pythonic import *

a, b, c = FPs("a b c", Float32())
rm = Const("rm", RNE().sort())
s = SolverFor('QF_FP')
```

```
s.add(Distinct(fpFMA(rm, a, b, c), fpAdd(rm, fpMul(rm, a, b),c)))
result = s.check()
m = s.model()
print(m)
print(f'fpFMA(rm, a, b, c) = {m.eval(fpFMA(rm, a, b, c))}')
print(f'fpAdd(rm, fpMul(rm, a, b),c) = {m.eval(fpAdd(rm, fpMul(rm, a, b),c))}')
```

The output gives the solution.

```
[a = -1.3333333730697632*(2**-1), b = -1.9999998807907104*(2**-1),
 c = -1.9999998807907104*(2**-1), rm = RTP()]
fpFMA(rm, a, b, c) = -1.333333134651184*(2**-2)
fpAdd(rm, fpMul(rm, a, b),c) = -1.3333330154418945*(2**-2)
```

Exercise 9. Modify the solution to Example 11 to show that floating-point addition is not associative, i.e., $a + (b + c) \neq (a + b) + c$.

4.7 Strings

It is often necessary to reason about string data when reasoning about programs. Reasoning about bit-vector representations of strings has the disadvantage that it requires fixing the string length up front. Also, the theory of bit-vectors does not include many of the utility functions for strings that exist in string libraries in programming languages. The SMT-LIB theory of strings provides support for variable-length strings and a large set of string operations. The quantifier-free logic name is `QF_S`. Typically, though, we use `QF_SLIA` since we need arithmetic to reason about string lengths.

Example 12. Given two strings, `x1` and `x2`, each consisting of no more than two characters, is it possible to build the string `"abbaabb"` using only 3 string concatenations (where each concatenation may use any previous result including `x1` and `x2`)?

We can solve this problem by building a *circuit* of string concatenations and using nondeterministic choice to pick the inputs for each concatenation.

```
from cvc5.pythonic import *
p, x, i = {}, {}, {}
for k in range(1, 13): p[k] = Bool("p" + str(k))
for k in range(1, 6): x[k] = String("x" + str(k))
for k in range(1,7): i[k] = String("i" + str(k))

result = StringVal("abbaabb")

s = SolverFor('QF_SLIA')
s.add(And(Length(x[1]) <= 2, Length(x[2]) <= 2))

s.add(i[1] == If(p[1], x[1], x[2]))
s.add(i[2] == If(p[2], x[1], x[2]))
s.add(x[3] == Concat(i[1], i[2]))

s.add(i[3] == If(p[3], x[1], If(p[4], x[2], x[3])))
s.add(i[4] == If(p[5], x[1], If(p[6], x[2], x[3])))
s.add(x[4] == Concat(i[3], i[4]))

s.add(i[5] == If(p[7], x[1], If(p[8], x[2], If(p[9], x[3], x[4]))))
s.add(i[6] == If(p[10], x[1], If(p[11], x[2], If(p[12], x[3], x[4]))))
```

```
s.add(x[5] == Concat(i[5], i[6]))

s.add(x[5] == result)

print(s.model() if s.check() == sat else "unsat")
```

Exercise 10. Use SMT to determine how many concatenations are needed to get "abbaabb" if x1 and x2 are both restricted to have a length of 1.

4.8 Quantifiers

We saw an example of quantified formulas in Example 3. Quantifiers can be enabled in SMT solvers by dropping QF from the logic name. However, enabling quantifiers typically increases the complexity of the decision problem significantly. In fact, solving UF problems is equivalent to solving the decision problem for first-order logic, Hilbert's original Entscheidungsproblem, which is undecidable. And although LIA, LRA, and NRA are decidable, the decision procedures are expensive. For these reasons, SMT solvers mostly handle quantifiers by attempting to find quantifier *instantiations* that, together with the other quantifier-free assertions, are unsatisfiable. For problems that are expected to be unsatisfiable, this approach can be quite effective. Moreover, by using different instantiation techniques and effort levels, a wide variety of problems can be solved.

cvc5 supports several techniques for handling quantified formulas, which can vary based on the logic. By default, cvc5 limits its effort so that it usually returns quickly with an answer of either unsat or unknown. For logics that include uninterpreted functions, it uses a combination of E-matching [31] and conflict-based instantiation [40]. In case the user wants to invest more effort, these techniques can be supplemented with techniques such as enumerative instantiation [38] (option enum-inst). For logics that admit quantifier elimination (e.g., quantified linear arithmetic or bit-vectors), it uses counterexample-guided quantifier instantiation [34,39], which is a complete procedure for these logics.

By default, cvc5 will generally not attempt to determine that an input with quantified formulas is satisfiable. However, more advanced techniques can be used to answer sat in the presence of quantified formulas, including finite model finding [37] (option finite-model-find), model-based quantifier instantiation [23] (option mbqi), and syntax-guided quantifier instantiation [35] (option sygus-inst).

In general, to set options that are not on by default, we can use the setOption solver method in Python, as shown below.

```
from cvc5.pythonic import *
s = SolverFor('UF')
s.setOption('enum-inst', True)
s.setOption('finite-model-find', True)
s.setOption('mbqi', True)
s.setOption('sygus-inst', True)
```

4.9 Non-standard Theories

cvc5 and Z3 support several theories that are not (yet) part of the SMT-LIB standard. We discuss a few of them briefly here, focusing on those supported by cvc5. More documentation about non-standard theories, including reference tables describing the supported operators can be found on the cvc5 website.

Sequences. The theory of *sequences* brings together features of the theories of arrays and strings. Similar to arrays, sequences are parameterized by the sort of their elements. So we can declare a sequence of integers, a sequence of bit-vectors, and so on. Like strings, sequences have a variable but finite length and can be concatenated together. The sequence theory is enabled whenever the string theory is enabled (e.g., by using the logic name `QF_S` or `QF_SLIA`). Note that Z3 also supports a theory of sequences that is mostly (but not entirely) compatible with the cvc5 version.

Example 13. Let `x` be a sequence of integers. Find a value for `x` such that the first and last elements sum to 9, and if we concatenate `x` with itself, then `(3,4,5)` appears as a subsequence.

```
from cvc5.pythonic import *
x, y, z = Consts("x y z", SeqSort(IntSort()))

s = SolverFor('QF_SLIA')

s.add(Length(x) > 0)
s.add(x[0] + x[Length(x) - 1] == 9)
s.add(y == Concat(x,x))
s.add(z == Concat(Unit(IntVal(3)), Unit(IntVal(4)), Unit(IntVal(5))))
s.add(Contains(y, z))

print(s.model() if s.check() == sat else "unsat")
```

Exercise 11. Show that it's not possible to have sequences `x`, `y`, and `z` such that `x` is a proper prefix of `y`, `y` is a proper prefix of `z`, and `z` is a proper prefix of `x`.

Finite Fields. cvc5 can reason about constraints over finite fields of order p, where p is any prime. It relies on the fact that a field of order p is isomorphic to the integers modulo p. The quantifier-free logic name for finite fields is `QF_FF`. At the time of writing, this theory is not supported by other SMT solvers.

Example 14. In a finite field of order 13, find two elements such that their sum and product are both equal to the multiplicative identity in the field.

Running this example requires a GPL build of cvc5, as explained in Sect. 1.

```
from cvc5.pythonic import *
F = FiniteFieldSort(13)
x, y = FiniteFieldElems("x y", F)
s = SolverFor("QF_FF")
s.add(x + y == 1)
s.add(x * y == 1)
print(s.model() if s.check() == sat else "unsat")
```

Exercise 12. In a finite field of order 13, find an element such that if you square it twice you get the multiplicative identity.

Finite Sets. cvc5 has support for the theory of finite sets. This theory supports basic set operations like membership, union, and intersection, as well as constraints on a set's cardinality. The quantifier-free logic name is `QF_FS`. At the time of writing, this theory is not supported by other SMT solvers.

Example 15. Verify that union distributes over intersection.

```
from cvc5.pythonic import *
S = DeclareSort("S")
A, B, C = [Set(i, S) for i in ["A","B","C"]]
s = SolverFor('QF_FS')
s.add(Not((A | (B & C)) == ((A | B) & (A | C))))
print(s.check())
```

Exercise 13. Does set difference distribute over intersection? If not, find a counterexample.

4.10 Combinations of Theories

So far, we have mostly seen examples of how to pose queries that involve a single theory. Part of the appeal of SMT solvers is their ability to mix reasoning about different theories. This can be done in a natural way. Any well-sorted formula is allowed, and all sort constructors can take any other sort as an argument.

One slight complication is the question of how to specify the logic name. It is always safe to use `ALL` as the logic name, though as mentioned above, it may be more efficient to give a more precise logic name. When mixing theories, cvc5 allows any logic name that follows the following rules. First, the logic name must start with the prefix `QF_` if the intent is to limit reasoning to quantifier-free formulas. The rest of the logic name can include any of the following components, in any order: (*i*) `A` for arrays; (*ii*) `UF` for uninterpreted functions; (*iii*) `BV` for bit-vectors; (iv) `FP` for floating-point numbers; (*v*) `DT` for datatypes; (vi) `S` for strings and sequences; (*vii*) either `IDL`, `RDL`, `LIA`, `LRA`, `LIRA`, `NIA`, `NRA`, or `NIRA` for arithmetic; (*viii*) `FF` for finite fields; and (*ix*) `FS` for finite sets. Thus, for example, `QF_AUFDTBVLRA` allows formulas that are quantifier-free and mix arrays, uninterpreted functions, datatypes, bit-vectors, and linear real arithmetic. Examples 10, 12, and 13 illustrate combinations of theories.

5 SMT Solver Outputs

As we have seen, the main result of an SMT query is either `sat` or `unsat`. In some cases, the solver may also output `unknown`. This can happen, for example, if the problem includes quantifiers. In this section, we discuss how to obtain more information from the solver in each case.

Satisfiable Queries. When a solver returns `sat`, we have already seen that one possible way to get more information is to call `get-model`, which returns values for all of the uninterpreted constants in the formula. A more fine-grained approach is to call `get-value` which takes a term as an argument and returns the value of that specific term.

Unsatisfiable Queries. When a solver returns `unsat`, it makes a quite strong statement: there is *no interpretation* of the user-declared symbols that satisfies the formula. SMT solvers can provide more information as to why a formula is unsatisfiable via an *unsat(isfiable) core*, a subset of the assertions that is already unsatisfiable. In SMT-LIB scripts, it can be obtained with the command `get-unsat-core`. The unsat core is not guaranteed to be minimal, but solvers generally make an effort to reduce its size as much as possible without having to solve additional SMT queries.

Some solvers can also produce *proofs* for the unsatisfiability of a formula, i.e., a structured argument showing how an inconsistency can be derived from an unsat core of the formula. A proof can serve as a certificate of the result and be used to independently validate the solver's response [4] . A proof (if supported) can be obtained in an SMT-LIB script with the command `get-proof`. The result is dependent on the proof system and format the solver uses to represent its reasoning. cvc5 has full support for proofs and unsat cores.

Consider again the Socrates example (Example 3). Below, we show how to retrieve an unsat core and a proof of its unsatisfiability.

```
from cvc5.pythonic import *

s = SolverFor('UF')

s.set("produce-proofs", "true")
s.set("proof-granularity", "theory-rewrite")
s.set("produce-unsat-cores", "true")

S = DeclareSort("S")
Human = Function("Human", S, BoolSort())
Mortal = Function("Mortal", S, BoolSort())
Socrates = Const("Socrates", S)

x = Const("x", S)

s.add(ForAll([x], Implies(Human(x), Mortal(x))))
s.add(Human(Socrates))
s.add(Not(Mortal(Socrates)))

print(s.check())
print("The core is: ", s.unsat_core())

p = s.proof()

print("The proof is:\n", p)
```

The first part of the output is the unsat core.

```
The core is:
   - (forall ((x S)) (=> (Human x) (Mortal x)))
   - (Human Socrates)
   - (not (Mortal Socrates))
```

The core contains all three assertions. In this case, the core is minimal, as all three are needed to derive `unsat`. The reasoning is shown in the proof. The result of the proof() method is a proof object which connects the input assertions to the conclusion (`unsat`) via a sequence of steps justified by proof rules. The proof rules used by cvc5 are documented on the cvc5 website.

Figure 1 shows a visualization of the proof as a tree. For readability, we use simple names to abbreviate long terms. Each node in the tree shows: (*i*) the formula proved (the conclusion); (*ii*) the name of the proof rule used; (*iii*) a numeric id; and (*iv*) the total number of descendants. Immediate children of each node represent premises required for the node's proof rule. The root of the tree is `let9`, which stands for `(not (and let4 let3 let2))`, where `let4`, `let3`, and `let2` represent the three assertions. This node has a single child containing the conclusion `false`, based on a proof tree whose leaves are the three assertions. The derivation of `false` depends on instantiating the quantified assertion (`let4`) with `x` as `Socrates`. This is done in node 5, only after `(forall ((x S)) (=> (Human x) (Mortal x)))` (i.e., `let4`) is rewritten (node 8) into `(forall ((x S)) (or (not (Human x)) (Mortal x)))` (i.e., `let8`). The instantiation `(or (not (Human Socrates)) (Mortal Socrates))` is named `let6`. Node 9 concludes `(not let6)` from the other assertions. Finally, node 2 concludes `false` from the mutually inconsistent clauses derived by the solver (where `let7` is `(not let6)`, `let2` is `(not let1)`, and `let5` is `(not let3)`).

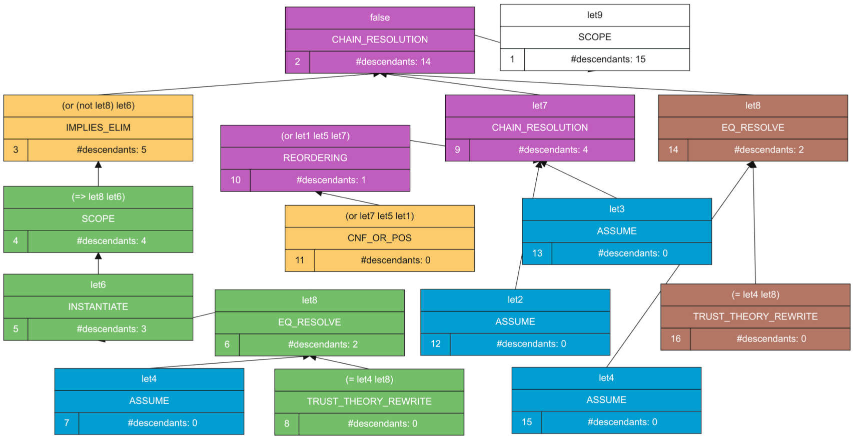


Fig. 1. A proof tree generated by cvc5

Unknown Queries. A solver returns `unknown` when it is unable to solve the input problem. There could be several different reasons for this. One is that the solver's procedure may be incomplete for the class of problems the input belongs to, which means that it is not always able to determine if the problem is satisfiable or not. Another possible reason is that some resource limit was exceeded, causing the solver to stop before it could find an answer. In SMT-LIB, the command `(get-info :reason-unknown)` can be used to request more information about why a solver returned `unknown`.

6 Conclusion

This tutorial is a basic introduction to using SMT solvers. There are numerous resources available for those who wish to learn more.

The SMT-LIB website smt-lib.org has details about the SMT-LIB standard [5], as well as links to software and an extensive collection of benchmarks. More information on the foundations of SMT and how solvers work under the hood can be found in several overview papers and book chapters [6,9,18]. There are also tool papers describing the most prominent SMT solvers, including: Alt-Ergo [15], Bitwuzla [33], cvc5 [29], MathSAT [14], OpenSMT2 [26], SMTInterpol [12], SMT-RAT [16], STP [22], veriT [10], Yices2 [19], and Z3 [32]. More information about cvc5 is available on its website.

Data Availability Statement. An artifact with all the examples and tools from the paper is available at: https://doi.org/10.5281/zenodo.12763927.

References

1. Abbott, J., Bigatti, A.M., Palezzato, E.: New in CoCoA-5.2.4 and CoCoALib-0.99600 for SC-square. In: Satisfiability Checking and Symbolic Computation. CEUR Workshop Proceedings, vol. 2189, pp. 88–94. CEUR-WS.org (2018). http://ceur-ws.org/Vol-2189/paper4.pdf
2. Ábrahám, E., Davenport, J.H., England, M., Kremer, G.: Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. J. Log. Algebraic Methods Program. **119**, 100633 (2021). https://doi.org/10.1016/J.JLAMP.2020.100633
3. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Barbosa, H., et al.: Generating and exploiting automated reasoning proof certificates. Commun. ACM **66**(10), 86–95 (2023). https://doi.org/10.1145/3587692
5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
6. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, chap. 33, pp. 825–885. IOS Press (2021)
7. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for a theory of inductive data types. J. Satisfiabil. Boolean Model. Comput. **3**, 21–46 (2007)

8. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theorie, Edinburgh, UK (2010)
9. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
10. Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 151–156. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_12
11. Brain, M., Tinelli, C., Rümmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. In: ARITH, pp. 160–167. IEEE (2015)
12. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31759-0_19
13. Church, A.: An unsolvable problem of elementary number theory. Am. J. Math. **58**(2), 345–363 (1936)
14. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
15. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories, Oxford, United Kingdom (2018). https://inria.hal.science/hal-01960203
16. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 360–368. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_26
17. Cotton, S., Maler, O.: Fast and flexible difference constraint propagation for DPLL(T). In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 170–183. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_19
18. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM **54**(9), 69–77 (2011). https://doi.org/10.1145/1995376.1995394
19. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
20. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_11
21. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press, Cambridge (1972)
22. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_52
23. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
24. Hilbert, D., Ackermann, W.: Grundzüge der theoretischen Logik, Berlin 1928. Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete **27** (1938)

25. Hodges, W.: A Shorter Model Theory. Cambridge University Press, Cambridge (1997)
26. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: an SMT solver for multi-core and cloud computing. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 547–553. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_35
27. IEEE: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008), pp. 1–84 (2019). https://doi.org/10.1109/IEEESTD.2019.8766229
28. Jr., H.S.W.: Hacker's Delight, 2nd edn. Pearson Education, Boston (2013). http://www.hackersdelight.org/
29. Kremer, G., Reynolds, A., Barrett, C.W., Tinelli, C.: Cooperating techniques for solving nonlinear real arithmetic in the cvc5 SMT solver (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13385, pp. 95–105. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-10769-6_7
30. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress, North-Holland, pp. 21–28 (1962)
31. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_13
32. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
33. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR arxiv:2006.01621 (2020)
34. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Solving quantified bit-vectors using invertibility conditions. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 236–255. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_16
35. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Syntax-guided quantifier instantiation. In: TACAS 2021. LNCS, vol. 12652, pp. 145–163. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_8
36. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_33
37. Reynolds, A., Tinelli, C., Goel, A., Krstić, S., Deters, M., Barrett, C.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 377–391. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_26
38. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 112–131. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_7
39. Reynolds, A., King, T., Kuncak, V.: Solving quantified linear arithmetic by counterexample-guided instantiation. Formal Methods Syst. Des. **51**(3), 500–532 (2017). https://doi.org/10.1007/s10703-017-0290-y
40. Reynolds, A., Tinelli, C., de Moura, L.M.: Finding conflicting instances of quantified formulas in SMT. In: Formal Methods in Computer-Aided Design, FMCAD

2014, Lausanne, Switzerland, 21–24 October 2014, pp. 195–202. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987613

41. Roselli, S.F., Bengtsson, K., Åkesson, K.: SMT solvers for job-shop scheduling problems: models comparison and performance evaluation. In: 14th IEEE International Conference on Automation Science and Engineering, CASE 2018, Munich, Germany, 20–24 August 2018, pp. 547–552. IEEE (2018). https://doi.org/10.1109/COASE.2018.8560344
42. Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. J. Math. **58**(345–363), 5 (1936)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

The Java Verification Tool KeY:A Tutorial

Bernhard Beckert[1], Richard Bubel[2](✉), Daniel Drodt[2], Reiner Hähnle[2], Florian Lanzinger[1], Wolfram Pfeifer[1], Mattias Ulbrich[1], and Alexander Weigl[1]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
`{beckert,lanzinger,wolfram.pfeifer,ulbrich,weigl}@kit.edu`
[2] Technische Universität Darmstadt, Darmstadt, Germany
`{richard.bubel,daniel.drodt,reiner.haehnle}@tu-darmstadt.de`

Abstract. The KeY tool is a state-of-the-art deductive program verifier for the Java language. Its verification engine is based on a sequent calculus for dynamic logic, realizing forward symbolic execution of the target program, whereby all symbolic paths through a program are explored. Method contracts make verification scalable. KeY combines auto-active and fine-grained proof interaction, which is possible both at the level of the verification target and its specification, as well as at the level of proof rules and program logic. This makes KeY well-suited for teaching program verification, but also permits proof debugging at the source code level. The latter made it possible to verify some of the most complex Java code to date. The article provides a self-contained introduction to the working principles and the practical usage of KeY for anyone with basic knowledge in logic and formal methods.

Keywords: Program verification · Deductive verification · Dynamic Logic · Java Modeling Language

"...and the aeroplane shot further away and again,
in a fresh space of sky, began writing a K, an E, a Y perhaps?"
—Virginia Woolf, *Mrs. Dalloway*

1 Introduction

What Is KeY? The KeY tool [3,4,15] is a state-of-the-art program verification tool for one of the most widely used programming languages: Java. Its capabilities enable the formal specification and verification of unmodified industrial Java code at source-code level. Notable examples of its application include the TimSort effort [28] and, more recently, the verification of a Java implementation of in-place super scalar sample sort [8], one of the fastest general-purpose sorting algorithms [18]. In addition to its role as a program verifier, KeY serves as a versatile research platform for implementing various formal methods for

© The Author(s) 2025

Java by leveraging KeY's symbolic execution engine. For instance, KeY has been used to facilitate the generation of test cases with high code coverage [6] and to implement a symbolic-state debugger [41]. The maturity of KeY's verification approach and of the tool make it suitable for teaching in BSc- and MSc-level courses. KeY is an academic, noncommercial tool that can be used freely by anyone (it is published under GNU Public License V2). It is completely written in Java, so it can run on any platform for which a Java virtual machine is available.

The roots of the KeY project trace back to 1999, when the continuous development and refinement of KeY and its verification methodology was started. On the occasion of KeY's 25th birthday, this tutorial serves to showcase the mature program verification and analysis tool that KeY is today.

This Tutorial, Its Accompanying Material, and Further Reading. This tutorial caters to all who want to learn about the KeY tool methodology: Newcomers to the field as well as experienced researchers in formal methods outside of deductive verification. It offers an exploration of KeY's underlying methodology, its capabilities, and its practical application. The tutorial covers the basics of KeY while giving a glimpse at its advanced features. Participants of the live conference tutorial gain hands-on experience with KeY. By the end of the tutorial, you are able to actively use KeY for (simple) verification tasks and to understand which advanced KeY features permit the verification of complex algorithms.

Videos, slides, and all examples of the conference tutorial, as well as the KeY tool itself, are available for download at www.key-project.org/tutorial-fm-2024. For further reading, the book on the KeY system, published in 2016 [4], contains tutorial chapters on using the KeY tool, based on both simple (chapter "Using the KeY Prover" [7]) and more advanced (chapter "Formal Verification with KeY: A Tutorial" [13]) examples.

KeY's Verification Methodology in a Nutshell. KeY's deductive verification engine is based on a sequent calculus for Java Dynamic Logic [17] (see Sect. 2). The calculus rules perform symbolic execution whereby all symbolic paths through a program are explored. Method contracts make verification scalable because one can prove one method at a time to be correct relative to its contract. Contracts do not need to be expressed in Dynamic Logic, but can be given at the source code level as *Java Modeling Language* (JML) annotations [48].

KeY features a domain-specific textual language (called *taclets*) to add axioms of theories and lemmas, and to define proof rules. This allows one to extend and tailor the deduction engine without having to know implementation details.

KeY's Interaction Patterns and User Interface. In contrast to most other program verification tools, KeY seamlessly combines *auto-active* interaction and *fine-grained* interaction: Interaction is possible both at the level of the Java verification target and its JML specification (auto-active interaction pattern), as well as at the level of proof rules and the underlying program logic (fine-grained interaction pattern). In auto-active verification (used in, for example, Why3 [22], Frama-C [47], AutoProof [63], VeriFast [44], VerCors [21], VCC [27], and Dafny [50]), interaction at the input level (adding, removing, or rephrasing specifications, adding hints) is user-friendly as it does not require knowledge and understanding of intermediate proof states. But the lack of insight into the intermediate proof steps makes it hard to identify which additional specification annotations might be needed or which need to be rephrased. On the other hand, fine-grained interaction (most popular with general purpose theorem provers such as Isabelle [53] or Coq [20]), where the proof is constructed either manually or with the help of proof scripts (which may use automatic proof tactics), can give deep insights into possible issues and provides effective control as the user can inspect partial proofs, but it requires a considerable amount of expertise.

Upon loading an annotated Java file proof obligations are automatically translated into Java Dynamic Logic and presented in the GUI. This GUI is a central component of KeY. Its design is based on a point-and-click interaction style to support proof exploration and proof construction. For instance, the selection of a calculus rule—out of over 1500(!)—is greatly simplified by allowing the user to highlight any syntactical subentity of the proof goal simply by positioning the mouse; a dynamic context menu offers only the few proof rules which apply to this entity. Drag-and-drop mechanisms greatly simplify the instantiation of quantified variables. Other supported interactions are the inspection of proof trees, the pruning of branches, and unlimited undoing of proof steps.

KeY's Verification Process. The user provides Java source code with annotations written in JML. These consist of requirement specifications as well as auxiliary specifications such as loop invariants and contracts of called methods. KeY translates these into proof obligations in Java Dynamic Logic. Now, the user is left with the choice of trying to let the prover verify fully automatically or of starting interactively by applying calculus rules to the proof obligation. If the user chooses to start the automated proof search strategy offered by the prover, the result can be either that (i) the prover succeeds in finding a proof or (ii) the prover stops after a number of steps with a partial proof. This is the point in the proof process, where the user gets involved. The user inspects the proof state and decides whether to continue with fine-grained interaction or to continue in auto-active style and revise the JML annotation or the source code. Recently, a lightweight proof scripting language was provided that complements the GUI's point-and-click style interaction. It fosters proof reuse and mitigates the need to redo the initial part of failed proof attempts by hand [18].

2 Verification Approach

Arguably, the most important structuring concept in programming languages is *methods* (aka procedures, functions, etc.). Methods *abbreviate* code that would else have to be repeated many times over, they *structure* a program into groups of related code, they foster *abstraction* of different behavior into a common implementation, and finally, they *encapsulate* the effect of a computation to local memory.

With methods being such a central structuring concept of programs, it is natural that verification proofs should reflect and be able to benefit from the structure inherent to the program under verification. In fact, without *structural similarity* between the verification target and the proof object, it is exceedingly difficult to verify complex software systems. This is why most modern deductive verification approaches, including KeY, are *contract-based* [37].

2.1 The Principle of Contract-Based Verification

A contract, in the context of software verification, is a specification artifact that makes it possible to mimic the method structure of a program in a correctness proof. The central idea is to describe the effect of a possible execution of a given method in terms of logical formulas. This has a decisive consequence: Whenever in a proof over a program, it is necessary to reason about a called method, then, instead of going into the implementation of that method, it is sufficient to consider the formulas in the method's contract. This simple and natural approach has dramatic consequences for program verification: (i) Since contracts consist not of code but of formulas, we replace program execution by *substitution* and *deduction*, and (ii) in contrast to code, which, in general, admits an unbounded number of different execution paths, contracts consist of a *finite* number of formulas whose semantics is declarative.

Together, these two observations enable *procedure-modular*, in some cases even linear-size [23], verification proofs: the structure of verification proofs follows the structure of methods, and an unbounded number of possible method executions is described with a finite number of logical formulas.[1] But how, exactly, are method contracts defined? And how does one ensure that a given method implementation conforms to its contract?

2.2 Method Contracts

A method contract, similar to a legal contract, has two main aspects: (i) It specifies the conditions under which it goes into effect and, if so, (ii) it gives guarantees. It may also (iii) specify collateral effects and give (iv) a temporal statute on delivery. Translated to the domain of programs and methods:

Definition 1 (Method Contract). *A* method contract *for a method* `m` *is a tuple* $(pre, post[, mod]\,[, acc]\,[, trm])$*, where pre and post are formulas, called* pre- *and* postcondition*, respectively, the optional* modifiers *mod and acc are set expressions over memory locations, and the optional* termination witness *trm is a first-order term of a type equipped with a well-order* $\prec$*.*

[1] A similar mechanism is *loop invariants* that describe an unbounded number of iterations. Indeed, procedure contracts can be seen to generalize loop invariants.

Its semantics *is as follows: if* `m` *is started in any execution state where pre holds, then, in any state where* `m` *terminates, post must hold as well. If mod is present, then* `m` *may change at most the value of memory locations in mod (otherwise, it may change anything). If acc is present, then* `m` *may read at most the value of memory locations in acc (otherwise, it may read anything). If trm is present, then its value must decrease with respect to* $\prec$ *before* `m` *is called recursively, thus enforcing termination of recursive calls. Otherwise,* `m` *may diverge.*

Just like legal contracts, method contracts interface with two parties: the client and the provider. In programs, the client is the code containing a call to `m`, while the provider is the implementor of `m`. It is the latter's responsibility to ensure *post* (and, if present, *mod*, *acc*, and *trm*) *provided that pre* holds at the time when `m` is called which is the caller's responsibility. The different parts of a contract lead to different verification tasks that may or may not be proven separately: *partial correctness*, i.e. the relation between *pre* and *post* established by the implementor, *framing*, i.e. the correctness of memory change (*mod*) and access (*acc*), as well as *termination*.

2.3 Java Modeling Language

So far, the components of contracts are left abstract. In deductive verification of imperative programs, one typically uses (typed) first-order formulas or expressions to formalize them and a program logic to express the semantics of contracts. However, languages such as Java are very much richer than first-order logic, which makes it tedious to write contracts. For this reason, it is common to specify contracts based on *behavioral modeling notations* having a rich syntax and thus being closer to the target language. Contracts written in such a modeling language are then automatically desugared into first-order and program logic [36]. In KeY we use the *Java Modeling Language* (JML) [48].

We introduce JML contracts by example. Listing 1 shows a contract for `binSearch()`, a recursive Java implementation of binary search between indices `low` (inclusive) and `up` (exclusive) on an integer array `a`:

We observe that JML is placed in Java comments augmented by a leading `@` sign (the remaining `@`'s are purely cosmetic). JML permits visibility modifiers ("`private`") with the same semantics as Java. Any side effect-free Java expression may occur in JML, any `boolean` expression can serve as a formula. Non-Java keywords in JML expressions are indicated by a leading backslash. Beyond Java expressions, first-order universal and existential quantifiers are allowed in JML. These are evaluated over the domain specified in their variable declaration and have an optional *range* expression, for example, `low <= idx < up`, which restricts the values being quantified over.

This idiom, where a quantifier ranges over integers and is further restricted by upper and lower bounds, is characteristic of specifications over array types.

The JML keyword `requires` indicates the *pre* slot of a contract. In the example, two `requires` clauses state the various index bounds and that array `a` is

```
/*@ private normal_behavior
  @   requires   0 <= low <= up <= a.length;
  @   requires   (\forall int x, y; 0 <= x < y < a.length; a[x] <= a[y]);
  @   ensures    \result == -1 || low <= \result < up;
  @   ensures    (\exists int idx; low <= idx < up; a[idx] == v) ?
  @               \result >= low && a[\result] == v : \result == -1;
  @   assignable \nothing;
  @   measured_by up - low;
  @*/
private int binSearch(int[] a, int v, int low, int up) {
  if (low < up) {
    int mid = low + ((up - low) / 2);
    if (v == a[mid])     { return mid; }
    else if (v < a[mid]) { return binSearch(a, v, low, mid); }
    else                 { return binSearch(a, v, mid + 1, up); }
  }
  return -1;
}
```

Listing 1. Recursive implementation of binary search with JML specification

sorted, respectively. If a keyword occurs multiple times, as here, then the conjunction of all expressions must hold. The JML keyword `ensures` indicates the *post* slot of a contract. The first `ensures` clause expresses that the returned value is either a valid index of `a` or `-1`. The keyword `\result` denotes the returned value of a method. The second `ensures` clause is a conditional expression (observe that quantifiers can be nested), saying that (i) either the searched element `v` is present in `a` between the given bounds, in which case a valid index, where `v` can be found, is returned or else (ii) the constant `-1` is returned.

The JML keyword `assignable` indicates the *mod* slot of a contract. A search method is expected not to change the heap, so we specify the empty set of locations, for which the keyword `\nothing` stands. To prove termination of the recursive implementation we add a `measured_by` clause. The top-level call will be of the form `binSearch(a,v,0,a.length)`, so initially the measure is equal to the value of `a.length`. It decreases at each call, because `low < low+((up-low)/2)+1` holds; it is never negative because the relation `low <= up` is maintained.

2.4 Dynamic Logic

How does one *prove*, for example, that the implementation of `binSearch` in Listing 1 conforms to its contract? In KeY we use dynamic logic, a program logic due to Pratt [55] (the name *dynamic logic* was coined in [39]).

In a nutshell, dynamic logic is obtained from Hoare logic [43] by closing it syntactically with respect to first-order formulas. In consequence, correctness assertions may be nested which adds useful expressiveness.

Definition 2 (Dynamic Logic, DL). *Dynamic logic extends first-order logic with a binary operator* $[p]\phi$ *and is inductively defined as follows: (1) Every first-order formula is a DL formula. (2) If* p *is a program and* ϕ *a DL formula, then* $[p]\phi$ *is a DL formula. (3) The set of DL formulas is closed under propositional and first-order operators.*

Formula $[p]\phi$ *is valid in a first-order model* $\mathcal{M}$ *if the following holds: For any execution state* s*, if* p *is started in* s *and it terminates in a state* s'*, then* ϕ *is valid in* $\mathcal{M}$ *and* s'*.*

Obviously, $[p]\phi$ expresses partial correctness of p with respect to postcondition ϕ, whenever ϕ is a first-order formula. A first-order contract $(\mathit{pre}, \mathit{post})$ for p can be expressed as $\mathit{pre} \rightarrow [p]\mathit{post}$, which corresponds to the *Hoare triple* $\{\mathit{pre}\}p\{\mathit{post}\}$ [43]. The remaining contract elements are discussed later.

DL being syntactically closed, we can define the dual operator $\langle\cdot\rangle$ by $\langle p\rangle\phi \equiv \neg[p]\neg\phi$, which expresses *total correctness* of p, keeping in mind $[p]\phi$ trivially holds for non-terminating programs and assuming programs are deterministic.

It is important to observe that DL is a *modal logic*: in general, executing p changes the execution state. However, it is convenient to assume the value of *first-order* variables stays invariant. For example, we might want to write $\forall x. (x \geq 0 \rightarrow [p](\texttt{\result} \geq x))$ and be sure that x is *not* changed by p. To achieve this, it is necessary, unlike in Hoare logic, to sharply differentiate between memory locations in programs (variables, arrays, fields, ...) and first-order variables. The former are modeled as *non-rigid* constants and functions whose interpretation may change from state to state; the latter are, as usual, evaluated under a *rigid* first-order model and variable assignment. In consequence, we do not permit quantification over program variables—this would result in extremely complex scoping rules. On the other hand, it makes perfect sense to write a DL formula such as $\texttt{i} \doteq 0 \rightarrow [p]\texttt{i} \not\doteq 0$ and expect it to be valid, for example, when p is `++i`.

For the KeY tool, we use a Java-specific extension of dynamic logic called JavaDL. The main difference to vanilla DL [38] is that JavaDL contains many predefined rigid and non-rigid first-order functions and predicates, including suitable first-order theories that model Java features in first-order logic. The intended first-order models $\mathcal{M}$ (Definition 2) must be defined accordingly. For example, there is a non-rigid function that returns the length of an array `a` in the current execution state as $\mathit{length}(\texttt{a})$. In JavaDL we also permit Java-style syntax `a.length`. JavaDL is a *typed* first-order logic whose type system includes all Java primitive and reference types that are equipped with Java's typing rules. Obviously, all JavaDL terms are assumed to be well-formed according to Java rules, which is enforced by KeY's parser.

2.5 State Updates

A common approach to performing logical inference in program logic is to compute the *weakest precondition* [32] of $[p]\mathit{post}$, i.e. the logically weakest formula $\mathit{wp}(p, \mathit{post})$ such that $\mathit{wp}(p, \mathit{post}) \rightarrow [p]\mathit{post}$ holds. It is constructed from post by unraveling p backwards. For example, in Hoare calculus $wp(x := e, \phi) = \phi[x/e]$ (assuming x is a scalar variable and e a simple expression). The weakest precondition computation is iterated until the beginning of a program p is reached. Branching statements split into several weakest preconditions, such that the overall result of the process is a finite set of first-order formulas $vc_1, \ldots, vc_n$ for which $\bigwedge_i vc_i \to [p]post$ holds. The vc_i are called *verification conditions* and can be discharged, for example, with automated theorem provers or SMT solvers. Iterative or recursive constructs require strongest invariants to compute wp; otherwise, (stronger) necessary conditions are obtained.

This *verification condition generation* (VCG) is simple and amenable to automation, but is problematic whenever full automation is not achievable:(i) Verification conditions tend to become large and complex, and then they are difficult to understand in case they are not provable; (ii) executing a program backwards is unnatural for humans and makes it hard to follow a failed verification attempt.

In KeY we assume that contracts and loop invariants (Sect. 3) for complex programs must be at least partially created manually. Getting them right requires understanding of intermediate proof situations. For this reason, the JavaDL inference system is not based on a VCG architecture but on *forward symbolic execution*. Unfortunately, computing the dual of $wp(p, post)$, i.e. the *strongest postcondition* of a program started in a state satisfying pre, is expensive and unnatural for assignment statements. Therefore, we use a technical trick that avoids computing explicit strongest postconditions:

Definition 3 (Elementary Update). *Let* `v` *be a program variable of primitive type and* `e` *a simple (not nested) and side effect-free expression such that the assignment* `v = e` *is well-formed. Let* e *be the first-order representation of* `e`.[2] *Then* $\texttt{v} := e$ *is called an* elementary update.

The semantics *of an elementary update* $\texttt{v} := e$ *are all state transitions where the value of* `v` *in the final state is set to the value that* e *had in the first state.*

Updates capture the effect of symbolic state changes and are streamlined to represent (simple) assignments. As we shall see below, updates can be viewed as explicit substitutions that represent a symbolic state change. By prefixing JavaDL formulas with updates, we can express that a formula is evaluated in the state represented by these updates:

Definition 4 (JavaDL with Updates). *If* u *is an update,* ϕ *a DL formula, and* e *a DL expression, then* $\{u\}\,\phi$ *is a DL formula and* $\{u\}\,e$ *a DL expression.*

2.6 A JavaDL Calculus

For a simple assignment `v = e`, the DL formulas $[\texttt{v = e;}\,p]\phi$ and $\{\texttt{v} := e\}\,[p]\phi$ are logically equivalent. This observation is the basis for a *forward symbolic*

2 From now on, we adopt the convention that the first-order translation of a Java expression uses the same letter but is typeset in Roman font.

execution calculus to prove the validity of JavaDL formulas: For each type of Java statement st in a program formula $[st;\, p]\phi$, we compute a finite set of formulas that implies $[st;\, p]\phi$ and, therefore, can *replace* it. These formulas have the form $\{\phi_i \rightarrow \mathcal{U}_i[st_i;\, p]\phi\}_i$, where $\mathcal{U}_i$ are updates, the st_i typically are (possibly empty) sub-statements of st, and the ϕ_i (optional) preconditions (the above replacement of an assignment is a special case of this general schema). This characterization permits to further reduce the $[st_i;\, p]\phi$ and so on. All that remains to do is to turn this schema into a calculus and to design the actual rules.

We assume the reader is familiar with the basics of sequent calculi (see, for example, [33]). As usual, we use naming conventions for schema variables: $\phi, \psi, \ldots$ stand for JavaDL formulas, Γ, Δ for sets of JavaDL formulas, and $\mathcal{U}$ denotes an arbitrary sequence of updates. More schema variables are introduced as needed. A typical (unary) rule may have the general form of the left rule schema below, where DL formulas ϕ, ψ are rewritten while the update $\mathcal{U}$ and formulas in Γ, Δ remain unchanged (Γ might contain assumptions or theories).

$$\frac{\Gamma, \mathcal{U}\phi' \implies \mathcal{U}\psi', \Delta}{\Gamma, \mathcal{U}\phi \implies \mathcal{U}\psi, \Delta} \qquad\qquad \frac{\phi' \implies \psi'}{\phi \implies \psi}$$

To make rule notation more succinct, we drop context formulas and leading updates in the following as in the rule schema above on the right, where contexts are *implicit*, but we *actually mean the rule above on the left*. With this convention in place, we formalize the observation at the beginning of this subsection as the sequent rule given below on the left. As usual in sequent calculi, the rules are applied bottom-up. On the right is the rule that stops symbolic execution once there is no further statement left to evaluate:

$$\frac{\implies \{\mathtt{v} := e\}\,[p]\phi}{\implies [\mathtt{v\ =\ e;}\, p]\phi}\ \text{assignment} \qquad\qquad \frac{\implies \phi}{\implies [\,]\phi}\ \text{emptyBox}$$

Example 1. Let us prove correctness of in-place value swap (ignoring possible arithmetic overflow) of two `int` variables `i`, `j`, as formalized in the sequent:

$$\mathtt{i} \doteq i_0,\, \mathtt{j} \doteq j_0 \implies [\mathtt{i\ =\ i+j;\ j\ =\ i-j;\ i\ =\ i-j;}](\mathtt{i} \doteq j_0 \land \mathtt{j} \doteq i_0)$$

After applying the assignment rule three times, then rule (emptyBox), we obtain:

$$\mathtt{i} \doteq i_0,\, \mathtt{j} \doteq j_0 \implies \{\mathtt{i} := \mathtt{i+j}\}\,\{\mathtt{j} := \mathtt{i-j}\}\,\{\mathtt{i} := \mathtt{i-j}\}\,(\mathtt{i} \doteq j_0 \land \mathtt{j} \doteq i_0) \quad (1)$$

The example shows that we need rules for applying an update to a first-order formula or term. Updates can be viewed as *explicit substitutions* [2], thus update application is obvious: A straightforward homomorphism on formulas and terms, except the base case: $\{\mathtt{v} := e\}\,\mathtt{w}$ yields e in case $\mathtt{v} = \mathtt{w}$ and $\mathtt{w}$, otherwise.

We apply the updates (1), starting with the last one, which yields (2) and, after two more update applications, the provable first-order sequent (3):

$$\mathtt{i} \doteq i_0,\, \mathtt{j} \doteq j_0 \implies \{\mathtt{i} := \mathtt{i+j}\}\,\{\mathtt{j} := \mathtt{i-j}\}\,(\mathtt{i-j} \doteq j_0 \land \mathtt{j} \doteq i_0) \quad (2)$$

$$\mathtt{i} \doteq i_0,\, \mathtt{j} \doteq j_0 \implies \mathtt{i+j-(i+j-j)} \doteq j_0 \land \mathtt{(i+j)-j} \doteq i_0 \quad (3)$$

At this point, three important observations can be made: (i) The first-order formula on the right of sequent (3) is the *weakest precondition* of the program and postcondition in Example 1. Updates allow us to compute it in a *forward* fashion. (ii) It is unnecessary to define a substitution operator on programs (which is highly complex for languages such as Java), because updates are applied only on formulas and terms. Difficulties, such as aliasing or side effects, are dealt with at the level of symbolic execution rules, as we shall see below. (iii) There is a potential inefficiency in the so-far *lazily* applied updates. For example, when some code is unreachable, that is discovered late. In addition, iterative substitutions can blow up term size drastically. The last point is mitigated by performing *eager* update simplification. To this end, we define *parallel* updates $\mathtt{v}_1 := e_1 \,||\, \cdots \,||\, \mathtt{v}_n := e_n$, where each slot is an elementary update and all $\mathtt{v}_i$ are different.

The semantics of parallel updates are those state transitions, where all the elementary updates are performed in parallel, i.e. the *old* values of the right hand side are used in each elementary update. For example, the parallel update $\mathtt{j} := \mathtt{i} \,||\, \mathtt{i} := \mathtt{j}$ simultaneously sets `i` to the previous value of `j` and vice versa. Since all left-hand sides in a parallel update are different, this is well-defined (for the moment, we ignore aliasing, which is discussed in Sect. 3). To turn a sequence of elementary updates into a parallel update, the following rewrite rule is applied, where u is any, possibly parallel, update:

$$\{u\}\,\{\mathtt{v} := e\} \rightsquigarrow \{u \backslash \mathtt{v} \,||\, \mathtt{v} := \{u\}\,e\} \qquad \text{seqToPar}$$

where update $u\backslash\mathtt{v}$ is identical to u, except elementary updates with left-hand side `v` are dropped. This is to keep left-hand sides unique and is justified by the fact that any `v` occurring in u on the left is overwritten by the later update of `v`.

If we apply rule (seqToPar) to the formula in sequent (1), we obtain

$$\{\mathtt{i} := \mathtt{i+j} \,||\, \mathtt{j} := \mathtt{(i+j)-j}\}\,\{\mathtt{i} := \mathtt{i-j}\}\,(\mathtt{i} \doteq j_0 \wedge \mathtt{j} \doteq i_0)\ .$$

Then, before applying rule (seqToPar) again, it is possible to perform arithmetic simplification on the expression `(i+j)-j`. Such a strategy of *eager* update parallelization and simplification helps to keep symbolic expressions small and is crucial for performance.

2.7 Forward Symbolic Execution of Straight-Line Programs

To be able to verify straight-line programs with the JavaDL calculus, two more components are needed: Handling complex expressions and conditional statements. We start with the former. Typically, in a rich language such as Java is that an array assignment could be of the form `e[e′] = e″`, where each of `e`, `e′`, `e″` might be a complex expression. Moreover, evaluation of `e″` can incur side effects that may or may not influence evaluation of `e′` (`i++` vs. `++i`). Symbolic execution must respect Java's evaluation rules and record side effects at the correct place. Not surprisingly, a large number of rules are required. Luckily, all of these rules follow the same simple principles. We discuss one typical representative. This rule handles the case when the array reference *nse* is not a simple expression and possibly has side effects.

$$\frac{\Longrightarrow [T_{nse}\ \texttt{v;}\ \texttt{v = }nse\texttt{;}\ \texttt{v[e] = e}'\texttt{;}\,p]\phi}{\Longrightarrow [nse\texttt{[e] = e}'\texttt{;}\,p]\phi}\ \text{assignmentUnfoldLeftArrayRef}$$

First, a fresh variable `v` is allocated that holds the reference expression *nse*. Subsequently, the original assignment is unfolded and *nse* replaced with `v`. The premise can now be symbolically executed, relying on `v` being simple. Of course, further rules must be applied to deal with `e`, `e`′.

All rules for complex assignments follow this simple schema:(i) Memorize a non-simple sub-expression, (ii) unfold a complex expression with the memorized value, (iii) arrange the sequence of assignments to reflect Java's evaluation rules.

The same principle is used to ensure that guards of conditionals and loops are side effect-free, simple expressions (here named *se*) before they are symbolically executed. In consequence, the rule for conditionals is straightforward:

$$\frac{se \doteq TRUE \Longrightarrow [p;r]\phi \qquad se \doteq FALSE \Longrightarrow [q;r]\phi}{\Longrightarrow [\texttt{if (}se\texttt{)}\ p\ \texttt{else}\ q;r]\phi}\ \text{ifElseSplit}$$

2.8 Procedure-Modular Verification: Contracts and Method Calls

To verify the example in Listing 1, we need to handle recursive procedure calls (for loops, see Sect. 3). We focus on a simple case to avoid the main idea getting buried under technicalities: Assume a method signature `static T m(T′ arg);` with a contract (*pre*, *post*). We design a JavaDL rule that, instead of inlining `m`'s implementation, uses its contract (how to verify contracts is shown next). In the conclusion of the rule below, we assign the result of a call to `m` with a simple argument `se` compatible to `T′` to a simple location expression `v` compatible to `T`. Further, we assume that *pre*′ and *post*′ are the desugared first-order translations of *pre* and *post*, respectively, where `res` corresponds to JML's `\result`.

$$\frac{\Longrightarrow \{\texttt{arg} := se\}\,pre' \qquad \Longrightarrow \{\texttt{arg} := se\,||\,\texttt{res} := c_r\}\,(post' \rightarrow \{\texttt{v} := \texttt{res}\}\,[p]\phi)}{\Longrightarrow [\texttt{v = m(se);}\,p]\phi} \tag{4}$$

The left premise validates that `m`'s contract goes into effect by proving the precondition with *se* as the value of `arg`. The right premise uses the postcondition in the remaining proof. To this end, first `res` is initialized with an unknown value (fresh Skolem constant c_r), then *post*′ is added as an assumption. Whatever *post*′ knows about `res` is propagated to `v` and can be used to establish $[p]\phi$.

The general case for method contract application can be more complicated. Specifically, for non-static method calls (dynamic dispatch), the implementation of `m` might be impossible to determine statically. In this case, the verification branches into different cases, one for each potential implementation. In addition, the caller expression must be correctly set up and possible side effects of the call, as described in the `assignable` clause, must be considered. Finally, `m` might terminate with an exception. We refer to [35] for a full treatment.

To formalize *verification* of a contract's correctness is easy in JavaDL, because the modal correctness formulas are closely aligned to the semantics of contracts. With the terminology from above, to verify a contract, we prove the following sequent (`arg` is the name of `m`'s parameter used in *pre*′):

$$\mathit{pre}' \implies [\texttt{res = m(arg);}]\mathit{post}'$$

To avoid a circular argument, rule (4) is *not* permitted, but `m` is *inlined.* Again, this does not yet account for the possibility that `m` may throw an exception. To exclude this case, one can simply wrap the method call in a `try` statement and add a check to the postcondition, ensuring no exception was thrown.

We close the section observing that the expressiveness of dynamic logic permits to formalize method contract correctness and method contract usage as a single JavaDL formula resp. a JavaDL calculus rule. This is in contrast to VCG style verification based on Hoare logic, where this must be encoded with numerous `assert` statements dispersed throughout the program under verification.

2.9 Proving the Contract of Binary Search

We prove the contract shown in Listing 1. Thus, we expect the following JavaDL formula to be provable:

$$\{\texttt{v} := v_0 \,||\, \texttt{low} := l \,||\, \texttt{up} := u\}\ (\mathit{pre}' \to \langle\texttt{res = binSearch(a,v,low,up);}\rangle\mathit{post}')$$

Observe that this is a *total correctness* formula while the rules so far were formulated with *partial correctness* operators. Fortunately, the calculus for partial and total correctness is exactly the same, except for Java constructs with potentially unbounded behavior. These are recursive calls and loops. To deal with the former, a check for the measure to decrease must be added to rule (4). When provable, total correctness of all method contracts in a given program implies total correctness of any program. This follows from a result proved in [51].

Before we can prove the DL formula above with KeY, there is one last loose end to tie up: It concerns how assignments involving array types are handled. Due to the considerations in Sect. 2.7 we can assume that all locations are simple and side effect-free. Yet the assignment rules—below the one for array access on the right of the sequent—are relatively complex:

$$\frac{\begin{array}{c}\texttt{a} \neq \texttt{null},\ 0 \leq e < \texttt{a.length} \implies \{\texttt{v} := \texttt{a}[e]\}\,[p]\phi \\ \texttt{a} \doteq \texttt{null} \implies [\texttt{throw new NullPointerException();}\,p]\phi \\ \texttt{a} \neq \texttt{null},\ 0 > e \vee e \geq \texttt{a.length} \implies [\texttt{throw new AIOoBException();}\,p]\phi\end{array}}{\implies [\texttt{v = a[e];}\,p]\phi}$$

This rule (as other array rules) reflects that in Java an array access can throw a `NullPointerException` or an `ArrayIndexOutOfBoundsException` (abbreviated with `AIOoBException`), which in general cannot be statically excluded (for symbolic execution of exceptions, see Sect. 3.3). The actual update happens in

```
/*@ private normal_behavior
  @   requires   (\exists int idx; 0 <= idx < a.length; a[idx] == v);
  @   requires   (\forall int x, y; 0 <= x < y < a.length; a[x] <= a[y]);
  @   ensures    0 <= \result < a.length;
  @   ensures    a[\result] == v;
  @   assignable \nothing;
  @ also private exceptional_behavior
  @   requires   !(\exists int idx; 0 <= idx < a.length; a[idx] == v);
  @   assignable \nothing;
  @   signals_only NoSuchElementException;
  @*/
private int binSearch(int[] a, int v) {
  int low = 0;
  int up = a.length;

  /*@ loop_invariant 0 <= low <= up <= a.length;
    @ loop_invariant (\forall int x; 0 <= x < low; a[x] != v);
    @ loop_invariant (\forall int x; up <= x < a.length; a[x] != v);
    @ assignable \nothing;
    @ decreases up - low;
    @*/
  while (low < up) {
    int mid = low + ((up - low) / 2);
    if (v == a[mid])        { return mid; }
    else if (v < a[mid])    { up = mid; }
    else                    { low = mid + 1; }
  }
  throw new NoSuchElementException();
}
```

Listing 2. Iterative implementation of binary search with JML specification

the first premise. Array updates $\mathtt{v} := \mathtt{a}[e]$ constitute a new class of elementary updates with a dedicated set of update application and simplification rules, reflecting the semantics of Java arrays. In particular, these rules take into account that in Java array references might be aliased.

3 Towards Real Java

So far, we learned how to specify and verify a simple program, but the preceding section left some gaps. The symbolic execution rules discussed above only consider updates on local variables without aliasing and, for the most part, without exceptional behavior. Furthermore, just like in Hoare calculus, KeY's JavaDL calculus requires loop invariants. In this section, we introduce the concepts necessary to specify and verify the iterative version of binary search in Listing 2: the heap model, exceptions and other abnormal termination, and loop invariants.

3.1 Aliasing: State Updates on the Heap

A major difficulty in verifying object-oriented programs is aliasing on the heap. Consider an assignment to a field `o.f`. Then, the assignment rule from Sect. 2 no longer suffices because changing the value of `o.f` might also change the value of `o2.f` if $\mathtt{o} \doteq \mathtt{o2}$. To accommodate aliasing, JavaDL models the heap as an array with indices (o, f) (called heap locations), where o is a first-order expression of type *Object* and f is a first-order expression of type *Field*, the type of field references. The axiomatization is based on the theory of arrays [52], but it is extended by axioms specific to JavaDL. The list of these axioms is found in [59], here we explain the functions defined through these axioms informally.

Given a program variable h of type *Heap*, a heap location (o, f), and an expression e, the expression $\mathrm{store}(h, o, f, e)$ evaluates to a heap identical to h except that the value of location (o, f) is e. For any Java type A, there is a function select_A such that $\mathrm{select}_A(h, o, f)$ evaluates either to the value at the location (o, f), if that value has type A, or to an underspecified value otherwise.

Now, we can give an update rule for field assignments. If either side of an assignment is a complex expression, we first apply unfolding rules similar to the rule (assignmentUnfoldLeftArrayRef) from Sect. 2.7. For a field assignment where both sides are simple expressions, we have the following rule. Similar to the rule seen in Sect. 2.9, we need a premise to deal with a possible `NullPointerException`. The first premise translates the assignment to an update using the store function on the heap.

$$\frac{\begin{array}{c}\mathtt{v} \not\doteq \mathtt{null} \implies \{\mathtt{heap} := \mathrm{store}_A(\mathtt{heap}, \mathtt{o}, \mathtt{f}, \mathtt{v})\}\,[p]\phi \\ \mathtt{v} \doteq \mathtt{null} \implies [\mathtt{throw\ new\ NullPointerException();}p]\phi\end{array}}{\implies [\mathtt{o.f\ =\ v;}p]\phi}\ \text{assignmentToField}$$

To support modular verification as presented in Sect. 2, we need a way to model the effects of a method call on the heap. KeY uses a variant of dynamic frames [45,60], an approach which uses sets of heap locations as first-class logical variables. To model the heap after a method call, we use a function which takes a heap h and a location set s and replaces the value of any location in s by an unknown value. This is accomplished by the anonymization function anon: The expression $\mathrm{anon}(h, s, h')$ evaluates to a heap equal to h except that all values of locations in s are taken from h'. If h' occurs nowhere else in the sequent, then these values are unknown. Then, exactly the information in the postcondition is what is known about the new values. Our anonymization is related to the "havoc" notion in Boogie [11].

3.2 Loop Invariants in JML and JavaDL

To verify unbounded loops, KeY requires a manually specified loop invariant. A loop invariant is a formula that holds before entering the loop and after every loop iteration. Additionally, we need a termination witness (called loop variant) to prove total correctness.

The loop invariant in Listing 2 consists of three clauses: The first limits the range of the index variables `low` and `up`, like in the precondition of the recursive version. The other two clauses differ from the recursive contract. The recursive contract states that the searched value is between the indices `low` and `up`. When using a loop invariant, we must instead state that the searched value is *not* between the indices `0` and `low` nor between `up` and `a.length`. (KeY also permits recursive loop contracts [64], but this is beyond the scope of this tutorial).

The loop variant (`decreases`) is an expression whose value is always at least `0` but strictly decreases with every loop iteration. Finally, the loop needs an `assignable` condition to prove the surrounding method's `assignable` condition.

When encountering a loop in JavaDL's calculus, one must prove three claims:(i) The loop invariant holds when entering the loop; (ii) the loop invariant is preserved by the loop body; (iii) after the loop terminates, the invariant ensures that the postcondition holds after executing the rest of the program. These claims are captured in the three premises of the following rule (a simplified version that only applies to loops without side effects in the loop guard and without abnormal termination; it also does not consider the loop variant):

$$\dfrac{\begin{array}{c}\Gamma \implies \mathcal{U} \mathit{inv} \\ \Gamma \implies \mathcal{U}\mathcal{A}((\mathit{inv} \land \texttt{cond} \doteq \mathrm{TRUE}) \rightarrow [\mathit{body}]\mathit{inv} \land \mathit{frame}) \\ \Gamma \implies \mathcal{U}\mathcal{A}((\mathit{inv} \land \texttt{cond} \doteq \mathrm{FALSE}) \rightarrow [\pi\omega]\phi)\end{array}}{\Gamma \implies \mathcal{U}\ [\pi\ \texttt{while (cond) \{}\ \mathit{body}\ \texttt{\}}\ \omega]\phi}\ \text{simpleInv}$$

Here, we drop the notational convention established in Sect. 2.6 and write the update $\mathcal{U}$ and antecedent Γ explicitly. We also write ω for the *rest of the program* and π for the *inactive prefix*, which may include a sequence of opening braces `{` and initial try blocks "`try {`". The *initial update* $\mathcal{U}$ captures the state of symbolic execution before the loop. The first premise ensures that the invariant *inv* holds upon entering the loop. The second and third premises contain the update $\mathcal{A} = \{\texttt{heap} := \mathrm{anon}(\texttt{heap}, \mathit{mod}, a_h) \,||\, \texttt{l}_1 := c_1 \,||\cdots||\, \texttt{l}_n := c_n\}$, Here, *mod* corresponds to the `assignable` clause, a_h is an unknown heap (i.e., a heap which occurs nowhere else in the sequent) and $\texttt{l}_i := c_i$ are updates which set any local variable $\texttt{l}_i$ written in the loop body to an unknown value c_i. The two updates $\mathcal{U}\mathcal{A}$ are applied sequentially to transfer that part of the symbolic state that is unchanged by the loop across the loop boundary. If, in that partially anonymized state, the invariant and loop guard both hold, executing the loop body must preserve the invariant and the *frame condition*, which ensures that any heap location outside *mod* is unchanged. If the invariant holds but the loop guard does not (the loop terminates), the postcondition must hold after executing the program rest ω.

3.3 Exceptions in JML and JavaDL

In Sect. 2 we considered programs that terminate normally. But the version of `binSearch` in Listing 2 throws an exception if the element is not found (instead of returning `-1`). To specify this, we add a second contract using the keyword `also`. That contract starts with `exceptional_behavior`, which specifies that the method terminates with an exception if the precondition holds. The keyword `signals_only` followed by a list of exception types states that the method throws no other exceptions except those listed.

The translation to JavaDL combines both contracts: The JavaDL precondition is the disjunction $\mathit{pre} \vee \mathit{pre}'$ of both preconditions, and the postcondition is

$$(\mathit{pre} \rightarrow \texttt{exc} \doteq \texttt{null} \wedge \mathit{post}) \wedge (\mathit{pre}' \rightarrow \mathrm{instanceOf}_{\texttt{NSEE}}(\texttt{exc}))$$

where *post* is the translation of the `ensures` clause and `exc` is a reserved program variable set when a `throw` statement is symbolically executed.

3.4 Integer Semantics

Recall that in Example 1, we glossed over the issue of arithmetic overflows. We treated Java's `int` type as the mathematical integers $\mathbb{Z}$ and all arithmetic operations on `int` as their mathematical counterpart (our discussion focuses on integers, but similar considerations apply to `byte`/`long`). Clearly, it is unsound to disregard overflows. Consider the DL formula

$$\texttt{i} \geq 0 \rightarrow [\texttt{i = i + 1;}](\texttt{i} > 0)$$

At first glance, it seems to be valid. But in case `i`'s value is the maximal `int` value, there is an overflow resulting in a negative value of `i`. To render the formula valid, we can strengthen the precondition by `i < Integer.MAX_VALUE`.

To permit flexibility in the choice of the arithmetic model, KeY translates operations `+`, `-`, `*`, etc., to *abstract* JavaDL functions during symbolic execution. For example, `a + b` becomes $\mathrm{javaAddInt}(\mathtt{a}, \mathtt{b})$ (assuming that `a`,`b` are of type `int`). The interpretation of these abstract functions can be configured in the KeY tool (option "intrules"). Three options for integer semantics are available:

(I) The default integer semantics, arithmeticSemanticsIgnoringOF, translates to $\mathbb{Z}$, as we did in Example 1. This semantics allows for easy prototyping and teaching—also specifications tend to be much simpler—but it is unsound. Nor is this semantics complete, as some valid formulas cannot be proven, such as $\texttt{i} \doteq \texttt{Integer.MAX\_VALUE} \rightarrow [\texttt{i = i + 1;}](\texttt{i} < 0)$. (II) To verify a program that does not rely on overflows, the semantics checkedOverflow is suitable. It checks that for all abstract functions, the result is in the value range of `int`, i.e. it proves the *absence* of overflows. While checkedOverflow is sound, it is not complete. If an intentional overflow occurs, the proof cannot be finished. Both proof and specification efforts tend to be bigger with this option than for the mathematical semantics. (III) The javaSemantics accurately models most operations on Java's `int` and provides soundness and completeness. All abstract functions are translated to accurate calculations for `int`, at the cost of even more complex proofs.

Integer semantics options let the user trade off the complexity of proofs and specifications against the accuracy of the modeling: Is an exact model of Java's `int` required, which will complicate the proof? Is showing the absence of overflows sufficient? Is the limited accuracy of mathematical integers acceptable? The answer will depend on the specific case.

Floating Point Numbers. KeY recently added support for floating point numbers, using a combination of theories in taclets and SMT solvers [1].

4 Inside KeY's Core

4.1 Prover Architecture

As discussed in Sect. 2.5, KeY does not have a VCG architecture. Unlike such tools as OpenJML [26] or Dafny [49], KeY comes with a built-in theorem prover, but can also use external SMT solvers. It works directly on Java source code avoiding an intermediate representation. Instead, it utilizes updates to achieve forward symbolic execution, relying on its JavaDL calculus and automatic prover to close goals. The latter is strong enough in many complex situations.

In addition to avoiding the limitations of VCG discussed in Sect. 2.5, this approach has four main advantages:(I) Proofs generated by KeY are self-contained without a reference to—or trust placed in—external tools. It is always possible to examine the current proof state in KeY without the need to understand, for example, the SMTLIB format [12]. (II) The user of the KeY prover and the tool itself work on the same structure and goals. This simplifies understanding of proofs, the underlying calculus, and potential errors. (III) The automation capabilities of KeY enable it to simplify any JavaDL formula, not only quantifier-free first-order expressions, during symbolic execution. Since the automation strategies aggressively simplify updates, first-order formulas, and terms while symbolically executing a program, many branches in a proof tree are closed early or are not created in the first place. Simplification is crucial to lessen the impact of *path explosion*—a well-known issue in symbolic execution [9]. (IV) KeY generates an explicit, self-contained *proof object.* A KeY proof can be saved and reloaded, even when it is incomplete. A proof consists of the claim to be proven plus a series of rule applications. This permits to share and re-play proofs, increasing trust in KeY artifacts and enabling reproducible results. Hence, the trusted code base is only KeY and its 25 years of experience.

The downside of the KeY architecture is that, when verifying exceptionally complex code, KeY's automatic capabilities may be insufficient. In this case, KeY can hand a (first-order) goal over to an SMT solver, such as Z3 [30] or cvc5 [10]. This is especially useful for floating point numbers (see Sect. 3.4). In this manner, KeY can still profit from the advances in SMT-solving technology, albeit at the cost of sacrificing self-contained proof objects.

4.2 Taclets

As a proof assistant, KeY allows significant flexibility regarding its underlying calculus. Most rules of the JavaDL calculus are not hard-coded but written in a simple, but expressive, language for such rules called *taclets*. We provide a succinct description of taclets. For a more in-depth coverage of taclets, their features, and correctness, see [57].

Taclets are very versatile and permit axiomatization of data structures, definition of symbolic execution rules, rules for propositional and first-order logic, etc. They allow users to define their own rules to accommodate a specific verification purpose. To ensure soundness of first-order taclets, KeY generates a proof obligation expressing the soundness of the taclet, which is proven in KeY itself.

We only present one form of taclets: *rewrite taclets*. Recall the rule in Sect. 2.6 for symbolically executing assignments. Listing 3 defines the same rule as taclet. The `assignment` rule has four parts: (i) A definition of *schema variables* matching formulas (`post`), program variables (`#loc`), and side effect-free expressions (`#se`); (ii) a `\find` clause, defining the formula "in focus," i.e., to be replaced in the premise—in this case a modality of any kind with an assignment; (iii) `\replacewith` providing the formulas in the premises; (iv) a `\heuristics` clause, instructing the automatic prover when this rule should be applied.

```
assignment {
  \formula post;  \program Variable #loc;  \program SimpleExpression #se;
  \find(\modality{#allmodal}{.. #loc = #se; ...}\endmodality(post))
  \replacewith({#loc:=#se}\modality{#allmodal}{.. ...}\endmodality(post))
  \heuristics(simplify_prog)
};
```

Listing 3. A taclet defining the rule for symbolically executing an assignment.

The opening and closing ellipses '`..`' and '`...`' in the modality stand for the inactive prefix π and the rest of the program ω, respectively (see Sect. 3.2).

5 Advanced Concepts for Object-Orientation

For the verification of non-trivial object-oriented programs, two specification features are important: (i) *Data abstraction* by which the content of data structures is represented using mathematical values thus hiding implementation details and (ii) *data encapsulation* that allows reasoning about data structures locally provided that any structure operates only on memory locations belonging to itself.

5.1 Ghost and Model Fields, Model Methods

Abstraction is relevant for programs operating on non-trivial data structures, as dealing with the details and memory layouts of data structure implementations unnecessarily increases proof complexity. So it is important (and often an enabling factor) to possess means to abstract from implementation details and to work with abstract values describing and capturing the state of data structure objects. For object-oriented programs, the state of an object is often best captured abstractly in form of one or more values in mathematical data types.

The canonical abstraction of the state of a doubly linked list implementation, for example, is a sequence of its entries. The expected behavior of list operations

```
interface List {
  //@ instance ghost \seq content;

  //@ requires 0 <= idx < content.length;
  //@ ensures \result == (int)content[i];
  int get(int idx);

  //@ ensures content == \old(content) + \seq(value);
  void add(int value);
}

class ArrayList implements List {
  int[] array;
  //@ invariant content == \array2seq(array);
  ...
}
```

Listing 4. Implementation and specification of a list with model entities

can be described in contracts using this abstraction. A client using such lists does not need to know anything about the data structure's actual implementation. KeY supports three means to introduce abstract values as JML annotations into class files: *Ghost fields*, *model fields*, and *model methods*.

Ghost fields (and variables) are fields (and variables) that only exist for verification purposes. Since JML annotations are written in comments, they are ignored during compilation. For verification, however, ghost entities are treated like normal Java fields and variables. In particular, ghost fields give rise to heap locations as outlined in Sect. 3.1. Ghost entities in JML may have types which are only available in JML but not in Java. In assignments, expressions that go beyond the expressiveness of Java (like quantifiers) can be used with ghost variables. Ghost fields and variables are often used to store redundant information or intermediate results, which are not required for computations at run time, but can considerably simplify deductive verification.

The example in Listing 4 illustrates how a ghost field is used to abstract from a concrete data structure. The `List` interface declares the ghost field `content` holding the list's abstraction, which is a sequence of values. The abstraction suffices to specify the contract of method `get`, which obtains the integer value stored at index `idx`. The method `add` ensures that a value is appended to the `content`. It is specified using the sequence operator "+" in KeY's JML. The implementing class `ArrayList` uses an array that actually holds the list's values. The connection between the abstract list and its implementation is established via a *coupling invariant*. In this case the function `\array2seq` can be used to read the sequence of values from an array.

Modifications of ghost fields must be made explicitly using assignments in contracts. For example, the contract of method `add` (not shown here, but available in the tutorial sources), must set `content` explicitly to the new value.

Model fields are, like ghost fields, only visible during verification and not at compile time. However, unlike ghost fields, model fields do not have a state of their own but are *observer symbols* whose value is computed from the current heap state. They are more like side effect-free Java query methods than Java fields. A model field is declared by adding the JML modifier `model`.

The benefit of model fields is that they need not (and cannot) be updated explicitly since they "automatically" change their value. However, verification of programs with model fields usually needs significantly more interactions than programs with ghost fields, and proofs tend to be larger and more complex.

Model methods are a generalization of model fields in the sense that they have arguments. They are side effect-free methods declared in JML annotations.

5.2 Dynamic Frames

Data encapsulation is closely related to data abstraction: If a data structure is well encapsulated, then its abstract value does not depend on memory areas outside the data structure. This is known as the *framing problem*: How to specify and verify that the abstract state of an object does not interfere with another unrelated object? Framing is usually addressed by requiring that the memory locations of data structures do not overlap. Over the last two decades, mainly three concepts to solve the framing problem have emerged: *Separation logic* [56], *ownership type systems* [31], and *dynamic frames* [46].

The KeY tool implements dynamic frames [60], where the set of locations that "belong" to a data structure, i.e. those locations that can be read or written by its operations are explicitly modeled as a set of memory locations, often called the *footprint* of an object. A ghost field is used to model this location set.

Revisiting the `List` example, in Listing 5 we specify at the *interface* level that the `get` query method may at most read memory locations in the footprint (using the keyword `accessible`). The function `add` may modify at most these locations (specified using `assignable`). When the footprint grows in `add`, only fresh locations that were not yet allocated prior to the call may be added to ensure that footprints remain separate. This is a typical specification pattern used when specifying and verifying object-oriented programs with dynamic frames. The `List` example is covered in the tutorial material.

```
interface List {
  //@ instance ghost \locset footprint;

  //@ accessible footprint;
  //@ assignable \nothing;
  int get(int index);

  //@ assignable footprint;
  //@ ensures \new_elems_fresh(footprint);
  void add(int);
}
```

Listing 5. Specification pattern using dynamic frames for the list interface

The value of a query invocation can only change if an element in the footprint is modified. The following axiom is available in KeY:

$$\big(\forall o, f.\ (o, f) \in \texttt{list.footprint} \rightarrow \mathit{select}(h_1, o, f) \doteq \mathit{select}(h_2, o, f)\big) \rightarrow \\ \texttt{get}(h_1, \texttt{list}, \mathit{idx}) \doteq \texttt{get}(h_2, \texttt{list}, \mathit{idx}) \quad (5)$$

It expresses that the `get` function computes the same result in heaps h_1, h_2 if all locations in `footprint` hold the same values in h_1, h_2. When lists are known to have disjoint footprints, then the dynamic frame axiom (5) allows to infer that adding an element to one list has no influence on a query to the other list.

6 KeY as a Tool for the Community

Due to its maturity and openness, KeY is a valuable tool for the community. This includes the use of KeY as a tool for verification projects or for teaching, but also the use of KeY in research projects for building new tools on top of it.

6.1 KeY as a Tool to Verify Real-World Software

Over the years, a plethora of case studies has been conducted, where KeY was used to verify a plethora of real-world algorithms and data structures. We present a selection; a more comprehensive list is on the KeY project website.

A verification case study that received much attention is TimSort, an algorithm combining merge and insertion sort. It is prominently used as Java's default for sorting collections of objects. However, that implementation had a bug and crashed for certain large collections. This issue was detected and explained in [29], a fixed version has been presented and verified with KeY in [28].

While the JDK uses TimSort to sort collections of objects, collections of primitive types are sorted using Dual Pivot Quicksort, which is a standard quicksort that partitions into three instead of into two parts. The implementation provided by the JDK has been proven correct in [19], which includes the sortedness property, the permutation property, and the absence of integer overflows.

In [24], the core of the JDK's Identity Hash Map was specified and verified. In that case study, a novelty is the use of several JML tools: KeY, the bounded model checker JJBMC [16], and OpenJML [25], to exploit the strengths of each of them and jointly verify a large project.

Researchers at CWI showed that Java's LinkedList implementation breaks when lists with more than 2^{31} elements are created [42]. They propose a fixed version and verified it successfully with KeY. This case study shows the capability of KeY to reason about bounded integer data types and handle overflows.

The most recent large case study performed with KeY is the verification of the sorting algorithm in-place super scalar sample sort [18]. This algorithm is efficient on modern machines, as it avoids branch mispredictions, allows high instruction parallelism by reducing data dependencies in the innermost loops, and it is very cache-efficient. This case study shows that with KeY it is possible to verify state-of-the-art sorting algorithms of considerable size (in this case about 900 lines of Java) and complexity *without having to modify the source code.*

6.2 KeY for Teaching

KeY is well suited for teaching. It comes with a GUI that provides context-specific actions, such as the rules that are applicable to the specific selected term. It provides means to inspect partial proofs and to explore the state of the prover interactively. The approach and the tool are very mature, and a lot of material exists that describes them in great detail (e.g., [4,5,14]). For these reasons, KeY is used in many courses at various universities, a list can be found at https://www.key-project.org/applications/key-for-teaching/. There is also a plethora of course notes and slides.

6.3 KeY as Library and Research Platform

In addition to the use of KeY as a standalone GUI-centric tool, it is possible to use KeY as a platform for research or to include it in a project as a library employing its symbolic execution and automated reasoning capabilities. One tool that uses KeY in such a way is CorC [58], which is an Eclipse-based tool that allows users to construct correct programs by stepwise refinement. To verify that the Java statements adhere to their "contracts" (pre- and postconditions created via refinement from the top-level specifications), CorC calls KeY as a backend.

KeYmaera [54] is an offspring of KeY that can be used to prove properties about cyber-physical systems, which are systems that have continuous behavior as well as discrete state changes (for example cars or planes). However, its successor KeYmaera X [34] is a green-field implementation and does not share a common code base with KeY anymore.

The Symbolic Execution Debugger [40] can be used to symbolically execute a program and obtain a tree of possible program paths. This helps to understand program and specification and to detect bugs, for example when unexpected paths are present or expected ones are missing. More recently, the Refinity tool [61] extends KeY by *abstract execution* [62] and lets one prove the correctness of refactorings. Both tools make use of KeY as a library.

6.4 Open Source and Open Development

KeY has been open source since the inception of the project in 1999. In February 2023 the sources were moved to a public repository on GitHub.[3] The open development model facilitates bug reports and feature requests. GitHub also provides the possibility to contact the developers.

The annual KeY Symposium takes place since 2002. With an international field of participants, it has been a breeding ground for new ideas and features for KeY. Growing over the years, the most recent edition has been the largest ever with about 40 attendees. To transfer knowledge from experienced to newer developers, two hackathons have been organized (in 2018 and 2024). Both events were a great success and led to multiple new features and bug fixes.

[3] https://github.com/KeYProject/key.

Acknowledgments. This work was supported by the DFG projects BE 2334/9-1, BU 2924/3-1, HA 2617/9-1, and UL 433/3-1 as well as the Helmholtz topic Engineering Secure Systems (KASTEL) and the Helmholtz pilot program KiKIT.

Data Availability Statement. All tools and demos of this tutorial paper are available at Zenodo (DOI: 10.5281/zenodo.11669182).

References

1. Abbasi, R., Schiffl, J., Darulova, E., Ulbrich, M., Ahrendt, W.: Deductive verification of floating-point java programs in KeY. In: TACAS 2021. LNCS, vol. 12652, pp. 242–261. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_13
2. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press (1996)
3. Ahrendt, W., et al.: The KeY tool: integrating object oriented design and formal verification. Software and System Modeling **4**(1), 32–54 (2005). https://doi.org/10.1007/s10270-004-0058-x
4. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book: From Theory to Practice. No. 10001 in LNCS, Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
5. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M. (eds.): Deductive Software Verification: Future Perspectives. No. 12345 in LNCS, Springer (2020). https://doi.org/10.1007/978-3-030-64354-6
6. Ahrendt, W., Gladisch, C., Herda, M.: Proof-based test case generation. In: Deductive Software Verification – The KeY Book. LNCS, vol. 10001, pp. 415–451. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6_12
7. Ahrendt, W., Grebing, S.: Using the KeY prover. In: Deductive Software Verification – The KeY Book. LNCS, vol. 10001, pp. 495–539. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6_15
8. Axtmann, M., Witt, S., Ferizovic, D., Sanders, P.: Engineering in-place (shared-memory) sorting algorithms. Comput. Res. Repository (CoRR) abs/2009.13569 (2020). https://doi.org/10.48550/arXiv.2009.13569
9. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. (CSUR) **51**(3), 50 (2018). https://doi.org/10.1145/3182657
10. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
11. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) FMCO, pp. 364–387. Springer, Berlin, Heidelberg (2006)
12. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB) (2016). www.SMT-LIB.org
13. Beckert, B., Hähnle, R., Hentschel, M., Schmitt, P.H.: Formal verification with KeY: a tutorial. In: Deductive Software Verification – The KeY Book. LNCS, vol. 10001, pp. 541–570. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6_16

14. Beckert, B., Hähnle, R., Schmitt, P. (eds.): Verification of Object-Oriented Software The KeY Approach. No. 4334 in LNCS, Springer (2006). https://doi.org/10.1007/978-3-540-69061-0
15. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69061-0
16. Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular verification of JML contracts using bounded model checking. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12476, pp. 60–80. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_4
17. Beckert, B., Klebanov, V., Weiß, B.: Dynamic logic for Java. In: Deductive Software Verification – The KeY Book. LNCS, vol. 10001, pp. 49–106. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6_3
18. Beckert, B., Sanders, P., Ulbrich, M.: Formally verifying an efficient sorter. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference TACAS, Luxembourg City, Luxembourg. LNCS, Springer, Cham (2024). https://doi.org/10.1007/978-3-031-57246-3_15
19. Beckert, B., Schiffl, J., Schmitt, P.H., Ulbrich, M.: Proving JDK's dual pivot quicksort correct. In: Paskevich, A., Wies, T. (eds.) VSTTE 2017. LNCS, vol. 10712, pp. 35–48. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_3
20. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07964-5
21. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 102–110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7
22. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, pp. 53–64. Wrocław, Poland (2011)
23. de Boer, F.S., Hiep, H.A.: Completeness and complexity of reasoning about call-by-value in Hoare logic. ACM Trans. Prog. Lang. Syst. **43**(4), 17:1–17:35 (2021). https://doi.org/10.1145/3477143
24. de Boer, M., de Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal specification and verification of JDK's identity hash map implementation. In: ter Beek, M.H., Monahan, R. (eds.) Integrated Formal Methods, pp. 45–62. no. 13274 in LNCS, Springer International Publishing, Cham (2022).https://doi.org/10.1007/978-3-031-07727-2_4
25. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35
26. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) 1st Workshop on Formal Integrated Development Environment, F-IDE, Grenoble, France, pp. 79–92. No. 149 in EPTCS (2014)
27. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: Contract-based modular verification of concurrent C. In: International Conference on Software Engineering – Companion Volume, pp. 429–430 (2009)

28. De Gouw, S., De Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's sort method for generic collections. J. Automated Reasoning **62**(6), 93–126 (2019)
29. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: the good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 273–289. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_16
30. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
31. Dietl, W., Müller, P.: Universes: lightweight ownership for JML. J. Object Technol. **4**(8), 5–32 (2005). https://doi.org/10.5381/JOT.2005.4.8.A1
32. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
33. Fitting, M.C.: First-Order Logic and Automated Theorem Proving, 2nd edn. Springer-Verlag, New York (1996). https://doi.org/10.1007/978-1-4612-2360-3
34. Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 527–538. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_36
35. Grahl, D., Bubel, R., Mostowski, W., Schmitt, P.H., Ulbrich, M., Weiß, B.: Modular specification and verification. In: Deductive Software Verification – The KeY Book. LNCS, vol. 10001, pp. 289–351. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6_9
36. Grahl, D., Ulbrich, M.: From specification to proof obligations. In: Deductive Software Verification – The KeY Book. LNCS, vol. 10001, pp. 243–287. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6_8
37. Hähnle, R., Huisman, M.: Deductive software verification: from pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 345–373. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_18
38. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (Oct, Foundations of Computing (2000)
39. Harel, D., Meyer, A.R., Pratt, V.R.: Computability and completeness in logics of programs (preliminary report). In: Hopcroft, J.E., Friedman, E.P., Harrison, M.A. (eds.) Proceedings of the 9th Annual ACM Symposium on Theory of Computing, Boulder, CO, USA, pp. 261–268. ACM, New York, NY (1977). https://doi.org/10.1145/800105.803416
40. Hentschel, M., Bubel, R., Hähnle, R.: Symbolic execution debugger (SED). In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification, 14th International Conference, RV, Toronto, Canada, pp. 255–262. No. 8734 in LNCS, Springer (2014). https://doi.org/10.1007/978-3-319-11164-3_21
41. Hentschel, M., Bubel, R., Hähnle, R.: The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging verification and More. STTT **21**(5), 485–513 (2018)
42. Hiep, H.-D.A., Maathuis, O., Bian, J., de Boer, F.S., van Eekelen, M., de Gouw, S.: Verifying OpenJDK's `LinkedList` using key. In: TACAS 2020. LNCS, vol. 12079, pp. 217–234. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_13
43. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. ACM **12**(10), 576–580, 583 (1969)

44. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven (2008). http://www.cs.kuleuven.be/~bartj/verifast/verifast.pdf
45. Kassios, I.T.: Dynamic frames: support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006). https://doi.org/10.1007/11813040_19
46. Kassios, I.T.: The dynamic frames theory. Formal Aspects Comput. **23**(3), 267–288 (2011). https://doi.org/10.1007/S00165-010-0152-5
47. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015)
48. Leavens, G.T., et al.: JML reference manual (2013). http://www.eecs.ucf.edu/~leavens/JML//OldReleases/jmlrefman.pdf, draft revision 2344
49. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
50. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. In: F-IDE 2014, pp. 3–15. No. 149 in EPTCS (2014)
51. Lidström, C., Gurov, D.: An abstract contract theory for programs with procedures. In: FASE 2021. LNCS, vol. 12649, pp. 152–171. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-71500-7_8
52. McCarthy, J.: Towards a mathematical science of computation. In: 2nd IFIP Congress, pp. 21–28. North-Holland (1962)
53. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
54. Platzer, A., Quesel, J.-D.: KeYmaera: a hybrid theorem prover for hybrid systems (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 171–178. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_15
55. Pratt, V.R.: Semantical considerations on Floyd-Hoare logic. In: 17th Annual Symposium on Foundations of Computer Science, Houston, TX, USA, pp. 109–121. IEEE Computer Society, Los Alamitos, CA (1976).https://doi.org/10.1109/SFCS.1976.27
56. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Symposium on Logic in Computer Science (LICS) 2002, pp. 55–74. IEEE Computer Society (2002).https://doi.org/10.1109/LICS.2002.1029817
57. Rümmer, P., Ulbrich, M.: Proof search with taclets. In: Deductive Software Verification – The KeY Book. LNCS, vol. 10001, pp. 107–147. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6_4
58. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D., Watson, B.W.: Tool support for correctness-by-construction. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 25–42. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_2
59. Schmitt, P.H.: First-order logic. In: Deductive Software Verification – The KeY Book. LNCS, vol. 10001, pp. 23–47. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6_2
60. Schmitt, P.H., Ulbrich, M., Weiß, B.: Dynamic frames in java dynamic logic. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 138–152. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18070-5_10

61. Steinhöfel, D.: REFINITY to model and prove program transformation rules. In: Oliveira, B.C.S. (ed.) APLAS 2020. LNCS, vol. 12470, pp. 311–319. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64437-6_16
62. Steinhöfel, D., Hähnle, R.: Schematic program proofs with abstract execution: theory and applications. J. Autom. Reason. **68**(7), 7:1–7:57 (2024)
63. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 566–580. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_53
64. Tuerk, T.: Local reasoning about while-loops. In: VSTTE Theory Workshop (VS-Theory) (2012)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

A Tutorial on Stream-Based Monitoring

Jan Baumeister, Bernd Finkbeiner, Florian Kohn(✉), and Frederik Scheerer

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{jan.baumeister,finkbeiner,florian.kohn, frederik.scheerer}@cispa.de

Abstract. Stream-based runtime monitoring frameworks are safety assurance tools that check the runtime behavior of a system against a formal specification. This tutorial provides a hands-on introduction to RTLola, a real-time monitoring toolkit for cyber-physical systems and networks. RTLola processes, evaluates, and aggregates streams of input data, such as sensor readings, and provides a real-time analysis in the form of comprehensive statistics and logical assessments of the system's health. RTLola has been applied successfully in monitoring autonomous systems such as unmanned aircraft. The tutorial guides the reader through the development of a stream-based specification for an autonomous drone observing other flying objects in its flight path. Each tutorial section provides an intuitive introduction, highlighting useful language features and specification patterns, and gives a more in-depth explanation of technical details for the advanced reader. Finally, we discuss how runtime monitors generated from RTLola specifications can be integrated into a variety of systems and discuss different monitoring applications.

Keywords: Monitoring · Specifications · Cyber-Physical Systems

1 Introduction

Runtime monitoring is an applied formal method that assures the safety of a running system by evaluating its behavior against a formal specification. In the stream-based approach, this specification is given in terms of equations that relate input streams, that contain raw data such as sensor readings, to output streams that transform and aggregate the incoming information. The values on the output streams are then checked against trigger conditions that indicate faulty or dangerous situations.

This work was partially supported by the Aviation Research Program LuFo of the German Federal Ministry for Economic Affairs and Energy as part of "Volocopter Sicherheitstechnologie zur robusten eVTOL Flugzustandsabsicherung durch formales Monitoring" (No. 20Q1963C), by the German Research Foundation (DFG) as part of TRR 248 (No. 389792660), and by the European Research Council (ERC) Grant HYPER (No. 101055412).

© The Author(s) 2025

This tutorial provides a comprehensive introduction to the RTLola monitoring framework. RTLola is the real-time extension [13] of the *Lola* specification language [8], which pioneered the stream-based approach. RTLola has been successfully applied to cyber-physical systems such as (unmanned) aircraft. Major case studies include the DLR ARTIS (Autonomous Rotorcraft Testbed for Intelligent Systems) family of aircraft developed at the German Aerospace Center (DLR) [4], and the fully-electric aircraft designed by Volocopter, a leading aircraft manufacturer of electric multi-rotor helicopters [2].

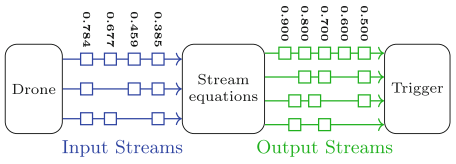
In the tutorial, we develop an RTLola specification for a real-world *detect and avoid* problem from the aerospace domain. We consider an autonomous drone flying in a shared airspace. The task of the monitor is to detect aircraft in the vicinity of the drone that might interfere with the drone's flight path. We will develop the specification in multiple steps, starting with the simple case of a single non-moving object. Our final specification will handle an apriori unbounded number of independently moving entities. Along the way, we introduce the relevant RTLola concepts and some fundamental background.

There are two possible ways to read this tutorial. If this is the reader's first encounter with RTLola, we recommend focussing on the development of the *detect and avoid* example. The tutorial starts in Sect. 2 with an overview of the monitoring framework and the running example. Afterwards, Sect. 3, Sect. 4, Sect. 5 and Sect. 6 extend the specification step-by-step, each section building up on the previous one. Finally, Sect. 7 explains how a monitor generated from a specification can be integrated into an existing system.

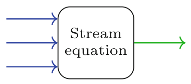
For readers interested in understanding the RTLola approach at a deeper technical level, the tutorial contains subsections with additional background. These subsections are marked as *for experts* to indicate that the subsections can safely be skipped at first reading. In this spirit, the *for experts* subsection of Sect. 2 provides a comprehensive overview of the various backends available in the RTLola framework; in Sect. 3, we discuss the static analysis of RTLola specifications. In Sect. 4, we introduce the type system, which is further refined in Sect. 5. In Sect. 6, we discuss finer points of parameterized specifications.

The tutorial is best experienced when following along in a browser window using the interactive RTLola Playground [16], which is available online [15]. Section 7 briefly explains how this tutorial is integrated into the Playground.

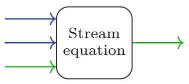
Related Work. There is a rich literature on runtime verification; we refer the reader to several introductory articles (cf. [1,11,19]). A previous tutorial on runtime monitoring has focussed on writing monitors using aspect-oriented programming [10]. Tutorials on stream-based monitoring have appeared as presentations at conferences (cf. [20,22]), but this is, to the best of our knowledge, the first tutorial paper that includes a hands-on development of a stream-based specification for a real-life application scenario. While the paper is based on the RTLola framework [13], the fundamental concepts apply in similar form to other stream-based monitoring approaches. Notable other stream-based monitoring approaches, in addition to Lola [8] and its successor Lola2.0 [12], are the Tessla [7] and Striver [18] tools.



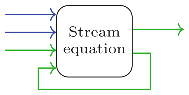
(a) The interaction of input and output streams.



(b) A stream equation over input streams.



(c) A stream equation including output streams.



(d) A stream equation over past stream values.

Fig. 1. An overview over stream-based runtime monitoring.

2 Overview

Runtime monitoring checks the behavior of a system, such as cyber-physical system (CPS), at runtime. The monitor receives input data from the system and analyses the data according to a formal specification. At each step, the monitor outputs if the specification is violated such that the system or an operator of the system can initiate countermeasures to return the system to a safe state. The next subsection introduces the general idea of stream-based monitoring followed by an introduction of the running example used in this tutorial.

Stream-Based Monitoring

Stream-based runtime monitoring interprets the system's input data as streams, i.e. a temporal discrete sequence over rich data values, transforms these input streams into output streams and expresses violations in the system's behavior. Figure 1a illustrates this idea: On the left side of the figure is the monitored system, which is in our case a drone. This system emits a sequence of timed values, which are called input streams. Stream equations then transform these input streams into output streams. Intuitively, stream equations are comparable to variable assignments in imperative programming languages. Yet, instead of only being evaluated once, like variable assignments, they are applied at every stream position to filter incoming data, compare values from different streams, or express complex temporal properties. Stream-based specification languages provide different stream access functions to reason about input streams (see Fig. 1b) and other output streams (see Fig. 1c). Stream equations can also access past values of a stream (see Fig. 1d) to express temporal properties. Finally, special boolean-valued output streams, called trigger streams, express violations in the system's behavior.

The next section introduces some syntax and semantics of stream equations in more detail, but first, we outline this tutorial's running example.

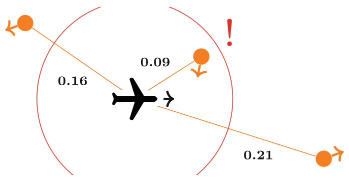


Fig. 2. A drone monitoring surrounding vehicles.

2.1 Running Example

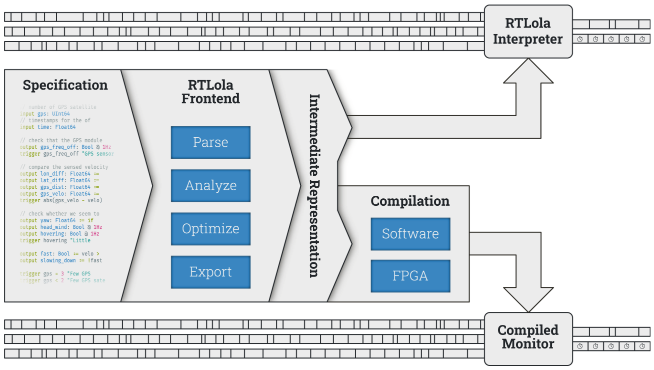
During this tutorial, we build a specification within the aerospace domain inspired by the *Detect and Avoid* specification introduced by Baumeister et al. [2]. More concretely, we consider an autonomous drone flying in a shared airspace and describe the detection of surrounding vehicles that might interfere with the drone's flight path. Figure 2 illustrates this property: The plane in the figure's center represents the drone's current position and the rays around the plane, the distance to other participants in the airspace. The exclamation mark indicates that this participant is close to the drone and is still approaching, so the drone should change the flight path to avoid a crash. From now on, we will call these participants *intruders* since they might interfere with the drone's flight path.

Algorithmically, the monitor should perform the following operations to monitor this scenario:

1. *Distance*: The distance to each intruder should be computed whenever the intruder or the drone moves.
2. *Closer*: To check if a specific intruder is approaching the drone, the monitor has to calculate whether the distance of the intruder is decreasing.
3. *Trigger*: If this approach continues over five seconds and the intruder is close to the drone, the monitor should notify the drone to initialize a counteraction.
4. *Stale*: The monitor should check if the intruder positions are regularly updated. Otherwise, it is declared out-of-range.

In this scenario, we expect the drone to provide the monitor with two sensor readings representing the input streams in our specification. First, we assume that the system has a global navigation satellite system (GNSS) to provide the monitor with the drone's latitude and longitude. Additionally, we assume that the system can detect other vehicles by sharing their position with other participants or by actively searching for intruders, e.g., with a radar, to get the latitude and longitude of the intruders. As output, the monitor provides the current distance to each intruder and the trigger if an approaching intruder is nearby.

The following sections explain different specifications describing these requirements in RTLola in a step-by-step fashion. We start with an intruder with a fixed position, e.g., a tree. Then, the specification is adjusted to handle a



```
// number of GPS satellite
input gps: UInt64
// timestamps for the of
input time: Float64

// check that the GPS module
output gps_freq_off: Bool @ 1Hz
trigger gps_freq_off "GPS sensor

// compare the sensed velocity
output lon_diff: Float64 :=
output lat_diff: Float64 :=
output gps_dist: Float64 :=
output gps_velo: Float64 :=
trigger abs(gps_velo - velo)

// check whether we seem to
output yaw: Float64 := if
output head_wind: Bool @ 1Hz
output hovering: Bool @ 1Hz
trigger hovering "Little

output fast: Bool := velo >
output slowing_down := !fast

trigger gps = 3 "Few GPS
trigger gps < 2 "Few GPS sate
```

Fig. 3. Overview of the RTLola framework.

moving intruder, e.g., a single moving drone. Finally, we adapt the specification to detect more than one approaching vehicle.

For Experts: Integration & Compilation

The RTLola specification language is embedded in an extensive framework to analyze and monitor specifications. Figure 3 provides an overview of this framework.

It is divided into the frontend and several backends. The frontend takes a specification file and produces an intermediate representation. This representation contains an abstract syntax tree of the specification annotated with additional information relevant to the backends. Additionally, the frontend optimizes the specification as presented in [3]. To verify the functional correctness of the specification, the intermediate representation can be inspected before executing the monitor, as shown in [9].

All backends have online and offline monitoring capabilities, i.e., they can monitor a system at runtime or monitor a log of its execution. RTLola specifications can be executed in the software-based interpreter [13] or compiled into the hardware description language VHDL [5] or imperative programming languages [17]. Providing both an interpretation and compilation ensures flexibility and efficiency: An interpretation allows for easy debugging and quick development times of the specification as it can easily be adjusted and reevaluated. A compilation, particularly a compilation to hardware, can provide highly optimized monitors that meet strict system requirements such as a low power consumption. The framework's versatility was confirmed in industrial case studies [2] with aerospace partners.

In the RTLola framework, the interpreter takes the specification as its intermediate representation and interprets it based on the incoming data from the system. It provides an extensive API to integrate it into existing implementations. Through the API, the interpreter can quickly adapt to different input data formats and sources.

The compilation takes the intermediate representation and produces executable code that implements a monitor for the given specification. For software, it produces code in an imperative programming language such as Rust. The hardware compiler produces VHDL code that can then be synthesized onto an FPGA. The monitor implementation receives inputs through input wires, and the current stream values are stored on the corresponding output wires or variables. After implementing the communication between the system and the monitor, it can be deployed with the system. Although this approach is less flexible than the interpretation, the resulting monitor is highly efficient once built and integrated.

3 Stream-Based Specifications

After Sect. 2 has introduced the general idea of streams and stream equations, this section presents the concrete syntax of RTLola and gives the first concrete examples. Output streams are declared using the `output` keyword followed by a stream expression describing the computation of a stream value. In comparison, input streams are declared using the `input` keyword and do not require a stream expression as their value is given by the system under observation. Trigger streams, special output streams with boolean stream expressions to convey violations to an operator, are defined using the `trigger` keyword.

Stream expressions consist of common arithmetic and logical operators such as addition, subtraction, and conjunction. Higher-level mathematical functions such as sine, cosine, or the square root can be enabled by importing the `math` module. To access past stream values at discrete positions RTLola includes the `offset(by: -n)` operator to access the n-th last value of a stream. As the n-th last value of a stream might not exist, as the stream, for example, only has n-1 positions yet, an offset operation must be followed by a default value to choose in that case.

Consider the following simplified specification of the scenario explained in Sect. 2. We limit ourselves to a single non-moving intruder instead of multiple moving intruders and assume a synchronous timing model, i.e., all streams are evaluated when the input streams receive new values.

Example 1 (A Static Intruder).

```
import math
input lat: Float
input lon: Float
constant intruder_lat: Float := 249.301
constant intruder_lon: Float := 23.453

output distance: Float := sqrt((intruder_lat - lat)**2.0 + (intruder_lon - lon)**2.0)
output closer: Bool := distance.offset(by: -1).defaults(to: distance) >= distance

trigger closer && distance < 0.1 "Too close to the intruder"
```

The two input streams `lat` and `lon` represent the measurements of the drone's GPS coordinates. The constants below capture the static position of the non-moving intruder. The output stream `distance` keeps track of the Euclidian distance between the drone and the intruder. To achieve this, it retrieves the current values of the input streams and uses the formula to compute the distance. The type of the `distance` stream is automatically inferred and can be omitted, denoted transparently in the example. This inferred type follows from the floating point numbers given by the stream accesses and the underlying functions that operate on this type. The output stream `closer` captures the temporal property that the drone gets closer to the intruder. For that, its stream expression compares the last value of the distance stream, expressed by the `offset`-operator, with the current one. Given that the stream expression consists of a comparison, the type of the `closer` stream is inferred as a boolean automatically. Finally, a trigger stream defines the condition when the distance is too close. To make the trigger more precise, we require that the `closer` stream also evaluates to `true`. Therefore, the trigger only activates if the drone moves towards the intruder.

3.1 Semantics

The semantics of an RTLola specification is defined as a relation between input and output streams. Intuitively, it compares every stream value at every time-point with the computed value described by the stream expression. The following definition gives the semantics [8,21] for a subset of RTLola to cover the general idea without focusing on concrete details.

Definition 1 (Simplified RTLola Semantics). *Let φ be an* RTLola *specification with input stream variables $i_1, ..., i_m$ and output and trigger stream variables $s_1, ..., s_n$. Let $\tau_1, ..., \tau_m$ be streams of length N of input values. The tuple $\langle \sigma_1, ..., \sigma_n \rangle$ of streams of length N is called an evaluation model with respect to $\tau_1, ..., \tau_m$ iff for each equation in φ the following holds:*

$$\sigma_i(j) = val(e_i)(j) \quad for \quad 0 \leq j \leq N$$

where e_i is the corresponding stream expression of s_i and $val(e)(j)$ is for the expressions in our example defined as:

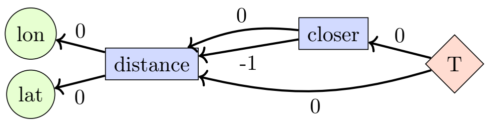
$$\begin{aligned} val(c)(j) &= c \\ val(i_t)(j) &= \tau_t(j) \\ val(s_t)(j) &= \sigma_t(j) \\ val(f(e_1, ..., e_k))(j) &= f(val(e_1)(j), ..., val(e_k)(j)) \\ val(e.offset(by{:}\ i).defaults(to{:}\ d))(j) &= \begin{cases} val(e)(j+i) & for\ 0 \leq j+i \\ val(d)(j) & otherwise \end{cases} \end{aligned}$$

3.2 Evaluation Algorithm

In contrast to imperative programs, the order of the equations in a stream-based specification does not influence the order of the evaluation. They are generally evaluated simultaneously, yet accesses between streams imply dependencies between streams and therefore affect the order. We represent these dependencies in a graph-based representation called the dependency graph. An analysis of this graph then computes a correct order of the stream evaluation: Every topological order of the dependency graph represents a valid order to process the streams during a single evaluation cycle.

Definition 2 (Dependency Graph). *Let ϕ be an* RTLola *specification. The dependency graph of ϕ is a directed weighted multi-graph $G = \langle V, E\rangle$ with $V = \{i_1, ..., i_m, s_1, ..., s_n\}$. An edge $e = \langle s_i, s_k, w\rangle$ is in E iff the expression of s_i contains s_k.offset(by: w) as a sub-expression. Analogously, edges with weight 0 are added for non-offset accesses.*

Example 2 (Dependency Graph). The following graph describes the dependency graph for the specification in Example 1:



Every node in the graph corresponds to a stream in the specification. For a better illustration, we mark input streams green, output streams blue, and trigger streams red. The input streams `lat` and `lon` do not have outgoing edges since input streams represent the input data and do not have a stream expression. The output stream `distance` accesses the current value of the `lon` and `lat` stream in the computation resulting in the 0-edge in the dependency graph. The `closer` stream accesses the current and last value of the `distance`-stream, resulting in a zero and an offset edge. Unlike input streams, trigger streams have no incoming edges since no stream expression can access these streams. In our example, the outgoing edges of the trigger are the accesses to the `distance` and `closer` stream.

Using this graph, we can compute different evaluation orders for this specification ensuring that the monitor accesses the intended stream values: The specification allows every order in which the `distance`-stream is evaluated after the inputs, the `closer`-stream after the `distance`-stream, and at the end of the evaluation the trigger stream.

For Experts: Static Analysis

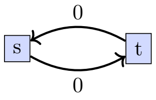
This expert subsection presents two static analyses based on the dependency graphs that guarantee a safe evaluation of stream-based specifications. The first analysis guarantees the existence of a unique evaluation model, whereas the second analysis computes an upper bound of the required memory. The latter analysis can determine if the monitor can run in a resource-constrained environment before execution.

Well-Formed Specifications. With this analysis, we define a syntactic criterion to guarantee the existence of a unique evaluation model. In general, we cannot find a unique evaluation model, if we cannot determine a order of the stream evaluation. This problem corresponds to a cycle in the dependency graph. First, consider the following examples of syntactically valid specifications and their dependency graph illustrating the problem.

Example 3 (Ill-defined Specifications and their Dependency Graph).

```
output s := !t
output t := s
```

```
output s := t
output t := s
```


$$\begin{tikzcd} s \arrow[r, bend right, "0"'] & t \arrow[l, bend right, "0"'] \end{tikzcd}$$

According to the semantics, the specification on the left has no evaluation model, since we cannot find an assignment for `s` and `t` that satisfies both equations. Such specifications are not well-defined and should be rejected by the RTLola framework. In comparison, the specification on the right has multiple evaluation models as long as `s` and `t` are equal. However, for this specification, we again cannot compute a valid evaluation order. As a result, neither specification can be evaluated algorithmically.
Both specifications have the same graph with an edge from stream `s` to stream `t` and an edge in the other direction, resulting in a cycle. Because of this cycle, we cannot determine a valid evaluation order and should reject the specification. We can give a syntactic criterion for well-definedness called well-formedness based on the dependency graph of a specification [8]:

Definition 3 (Well-formedness). *A specification is well-formed, iff for every cycle in its dependency graph, the accumulated edge weight of the cycle is not zero.*

As described in the definition, we do not forbid every cyclic behavior. The next example shows a valid specification containing a cycle in the dependency graph. It sums all values of the input stream `a`:

Example 4 (Valid Cycle).

```
input a : Int
output sum := sum.offset(by: -1).defaults(to: 0) + a
```

Here, the `sum` uses the offset expression to access the past value of itself. This access results in a cycle in the dependency graph, but the sum of this cycle is negative. Intuitively, this is allowed since past values are already computed, and we can ignore these accesses when building the evaluation order.

Static Memory Bounds. Secondly, we can determine the amount of values that need to be kept in memory for every stream. Again, this analysis is based on the dependency graph and allows giving static memory bounds for specifications.

Definition 4 (Memory Bound). *Let $G = \langle V, E \rangle$ be the dependency graph of the specification φ. For every stream s in φ its memory bound is determined as $max(\{-w \mid \langle o', s, w \rangle \in E\})$.*

Intuitively, the memory-bound of a stream is defined by the largest offset at which the stream is accessed. All values before that bound are not required for further computation and can be discarded. For the specification in Example 1 the memory bound of all streams is zero except for the `distance` stream. Given that the stream is accessed with an offset of -1, the memory bound of this stream is one. The memory bound of the specification as a whole is then the sum over the memory bounds of all input and output streams.

4 Event-Based and Periodic Streams

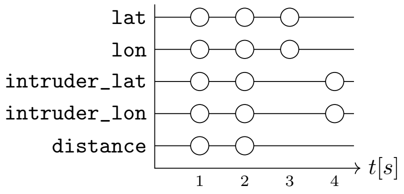
This section lifts the simplification of a static intruder to a moving intruder. For this, consider the following naïve extension of Example 1, where the intruder position is given as additional input streams `intruder_lat` and `intruder_lon`.

Example 5 (A Synchronous Moving Intruder).

```
import math
input lat: Float
input lon: Float
input intruder_lat: Float
input intruder_lon: Float

output distance :=
    sqrt((intruder_lat - lat)**2.0
        + (intruder_lon - lon)**2.0)
output closer := distance.offset(by: -1)
    .defaults(to: distance) >= distance

trigger closer && distance < 0.1
    "Too close to the intruder"
```



This specification is correct in a synchronous setting, i.e. a setting where all input streams receive a new value simultaneously. Yet, in reality, the intruder and the drone are independent systems. Hence, the measurements of the intruder's position might not be synchronous with the drone's position measurements. In an asynchronous setting, input streams receive values independent of each other. For this, every output stream and trigger is only evaluated if the input streams they (transitively) depend on receive a new value at the same time. Consider the specification from Example 5 in a synchronous and asynchronous setting as illustrated by the trace on the right. First, all values are received synchronously and the `distance` stream is computed. It might seem correct, yet all output streams and trigger streams still transitively or directly depend on all input streams. As a result, the specification is effectively synchronous again and behaves invalidly if not all inputs receive a new value at the same time, as depicted later in the trace. As we can see, if the current position and the intruder position are not synchronized, the `distance` stream is not updated as expected.

The next example depicts the corrected specification, where input streams are accessed asynchronously using hold-accesses:

Example 6 (A Moving Intruder).

```
import math
input lat: Float
input lon: Float
input intruder_lat: Float
input intruder_lon: Float

output distance @((intruder_lat && intruder_lon) || (lat && lon)) :=
      sqrt((intruder_lat.hold(or: 0.0) - lat.hold(or: 0.0))**2.0 +
      (intruder_lon.hold(or: 0.0) - lon.hold(or: 0.0))**2.0)
output closer @((intruder_lat && intruder_lon) || (lat && lon)) :=
    distance.offset(by: -1).defaults(to: distance) >= distance

trigger @1Hz closer.aggregate(over_exactly: 5s, using: forall).defaults(to: false) &&
      distance.hold(or: 1.0) < 0.1 "Too close to the intruder"
```

While the `distance` stream mathematically performs the same computation, a `hold()` lookup is used to avoid a direct dependency. This lookup refers to the most current available value of the accessed stream and does not require that the accessed value is computed at the same time. However, there might not be such a value when the accessing stream is evaluated, so a default value must be supplied similar to the offset operator. As the timing of the `distance` stream is decoupled from any input stream, it has to be explicitly specified when it should produce new values. Syntactically, this is specified through a positive boolean expression over input streams following the `@` after a stream's name. This expression is called the *activation condition* of the stream and symbolically describes the events at which the stream is evaluated. In the example, the `distance` stream is evaluated whenever the intruders *or* the drone's position changes. For the `closer` stream the activation condition is the same as for the `distance`, but is automatically inferred because of the synchronous access.

Moreover, the trigger in the specification has changed. It is now periodic, a concept which will be introduced in the subsequent section.

4.1 Periodic Streams

In reality, it is often required that a monitor not only reacts to system actions but can also proactively produce verdicts about the system's health. Otherwise, the monitor could not detect a frozen system because it would freeze as well. In RTLola, proactive monitoring is achieved through streams evaluated at a fixed frequency called periodic streams. A periodic stream can be specified by giving a frequency or period annotation like `1Hz` or `1s` after the `@` keyword. These frequencies are independent of input streams and to access input streams from periodic output streams we can either use `hold`-accesses or sliding windows. In our example, we use a sliding window in the trigger stream to make the specification more robust against GPS fluctuations.

Sliding window aggregations aggregate over every value in a given time frame of a stream using an aggregation function. More concretely, the window in our

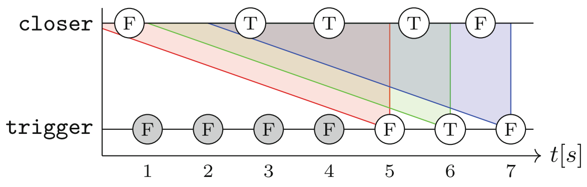


Fig. 4. The functioning of sliding windows exemplified based on Example 6

example evaluates only to true if every `closer` value in the last 5 s is true. The functionality of this sliding window is visualized in Fig. 4. In our specification, the trigger stream is evaluated with a fixed frequency of one second whereas the `closer` stream depends on the inputs. For the first four seconds, the window does not aggregate any values given that the whole duration is not available at the start of the monitoring. In this case, the window evaluates to the default value `false` instead. Afterwards, the window aggregates over different numbers of closer values as illustrated by the different colors.

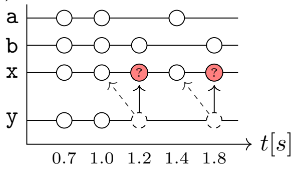
RTLola supports a set of aggregation functions as `exists`, `avg`, `sum`, `count`, `min` or `max`.

For Experts: RTLola's Type System

Similar to many programming languages, RTLola has a type system that ensures only valid specifications can be executed. The type-system of RTLola is twofold: The value type of a stream describes how the memory should be interpreted, i.e., as a signed or unsigned integer, a floating point number, or a string. The underlying value type system ensures that only compatible values are used in an operation, such that, for example, no string and number are added. The pacing type of a stream describes its timing behavior, i.e., it determines the points in time when the stream is evaluated. This pacing type system ensures that every direct access, also called synchronous access, is valid. Consider the following example specification with an invalid synchronous access:

Example 7 (Invalid Synchronous Access).

```
input a: Bool
input b: Bool
output x @a := a
output y @b := x
```



The specification on the left has two output streams: The stream `x`, evaluated whenever the input `a` receives a new value, with that same value of `a`. The stream `y` is evaluated whenever the input `b` receives a new value and takes the value of `x`. The diagram on the right exemplifies the timing behavior of the streams. At time 1.2 and 1.8, only stream `b` receives a new value. Consequently, the stream `x` is not computed, and with that stream `y` cannot access the value of `x` at these points in time. These timing errors can lead to invalid memory accesses at runtime, so the type checker rejects the specification. To fix the specification, one could use a `hold` access from `y` to `x` similar to Example 6, resulting in the greyed-out arrows in the diagram on the right.

The Pacing Type System. Intuitively, the pacing type system declares a synchronous access valid if the accessed stream is evaluated at least at the same points in time as the accessing stream. For non-periodic, also called event-based streams, this property can be checked by asserting logical implication between the activation condition of the accessing stream and the accessed stream. In Example 7, it is easy to see that $a \rightarrow b$ does not always hold, but, for example, $a \wedge b \rightarrow a$ does. A similar condition holds for periodic streams: A synchronous access between periodic streams is valid if the accessing stream runs at a slower pace than the accessed stream. Concretely, the period of the accessed stream has to be a multiple of the period of the accessing stream. In addition, synchronous accesses between periodic and event-based streams are never valid.

5 Stream Lifecycle

In this section, we optimize the specification from Example 6 by guarding computationally heavy operations with cheap-to-evaluate predicates. Concretely, we only compute the `distance` and `closer` streams and the trigger when the intruder is in range. These optimizations are enabled by lifting the assumption that a stream exists from the start to the end of the monitor. Instead, we allow for dynamic stream creation, i.e., a stream can be created and removed from the monitor at runtime. In the following, we refer to the creation of streams as their spawn behavior and their deletion as their close behavior. In between its spawn and close action, a stream is evaluated as defined by its stream expression.

Syntactically, the three steps of a stream's lifecycle are specified by three sub-clauses in the stream's definition:

```
output o
    spawn @p_s when c_s
    eval @p_e when c_e with e
    close @p_c when c_c
```

Each of these clauses can feature a pacing and a boolean stream expression preceded by the `when` keyword. The pacing of a clause *statically* determines when the clause could be evaluated, i.e., whenever event `a&b` occurs or at a frequency of `1Hz`. The stream expression preceded by `when` is evaluated at the time points described by the pacing. It constrains these time points dynamically based on runtime values, i.e., the action corresponding to the clause is only taken *when* this expression evaluates to true. The evaluation clause features a stream expression defining *how* the stream's value is computed after the `with` keyword.

As for the previous sections, pacings can be omitted and inferred by the type checker for brevity. The `when` expression of a clause can also be omitted, representing the constant `true`. If a stream's spawn or close clause is omitted, the stream exists from the start or until the end of the monitor, respectively. If only an `eval with` clause exists, we can use the short-hand notation `:=` as used in the previous sections.

Consider the following extension of Example 6 in which every stream in the specification is now composed of three clauses: a spawn clause, an eval clause, and a close clause.

Example 8 (An Out-of-Range Intruder).

```
import math
input intruder_lat: Float
input intruder_lon: Float
input lat: Float
input lon: Float

output distance
   spawn @(intruder_lat && intruder_lon)
   eval @((intruder_lat && intruder_lon) || (lat && lon))
      with sqrt((intruder_lat.hold(or: 0.0) - lat.hold(or: 0.0))**2.0 +
           (intruder_lon.hold(or: 0.0) - lon.hold(or: 0.0))**2.0)
   close @true when stale.hold(or: false)
output closer
   spawn @(intruder_lat && intruder_lon)
   eval with distance.offset(by: -1).defaults(to: distance) >= distance
   close @true when stale.hold(or: false)
output stale
   spawn @(intruder_lat && intruder_lon)
   eval @10s with intruder_lat.aggregate(over: 10s, using: count) = 0
   close @true when stale.hold(or: false)

trigger
   spawn @((intruder_lat && intruder_lon) || (lat && lon))
      when distance.hold(or: 1.0) < 0.1
   eval @1Hz when closer.aggregate(over_exactly: 5s, using: forall).defaults(to: false)
        with "Intruder detected"
   close @true when stale.hold(or: false)
```

Another notable change is the added `stale` stream. It defines when the intruder is considered out of range by checking for any GPS coordinate updates in the last 10 s. The `spawn` clause of the stream defines that this condition is only monitored once a GPS location is received from the intruder. The omitted `when` expression in the spawn clause is equivalent to a `when true` definition. The `close` clause of the stream defines that the condition should no longer be monitored as soon as it becomes true for the first time. However, the stream is spawned again if a new intruder GPS location reactivates the spawn condition of the stream.

The `distance` and `closer` streams inherit the same `spawn` and `close` clauses as the `stale` stream, further excluding redundant computations. The trigger also inherits the same close condition as the other two streams. Yet, its spawn condition differs slightly. The `distance.hold(or: 1.0)< 0.1` expression is moved from the main trigger condition to its spawn condition, adapting the spawn activation condition accordingly. That way, the computationally heavy sliding window aggregation is only performed once the intruder is close enough.

Besides optimizing specifications, dynamic stream creations increase the expressiveness of the RTLola specification language as shown in the next subsection.

5.1 Deadline Watchdogs

Deadline watchdogs are common specification requirements in CPS and can be expressed in natural language as follows: "t seconds after event e a condition c must hold." Such requirements can be represented in RTLola using dynamically created streams:

Example 9 (A Deadline Watchdog).

```
output timer
   spawn when e
   eval @ts with true
   close when timer

trigger
   spawn when e
   eval @ts when !c with "The deadline was missed"
   close when timer
```

For the watchdog, we define a new stream `timer` that is spawned with the start of the watchdog, i.e. the event e spawns the watchdog. This is the starting point of the annotated frequency, so this stream is evaluated for the first time t seconds after e. Since this value is immediately true, we also close the stream after these t seconds. The trigger stream has the same `spawn` and `close` condition and is evaluated with the same frequency. Here, we check if the condition c is satisfied and use the `timer` stream as a helper function to immediately close the trigger stream after the first evaluation.

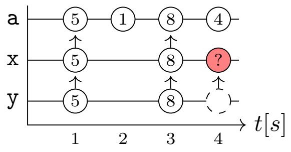
For Experts: Semantic Types

The introduced `when`-conditions further refine the timing of a stream and add another point of failure for synchronous accesses. This problem is solved with another type system.

Semantic Types and Event-Based Streams. The following example illustrates a possible point of failure using synchronous accesses and `when`-conditions:

Example 10 (Invalid Semantic Types).

```
input a: Int
output x
   eval @a when a > 4 with a
output y
   eval @a when a > 3 with x
```



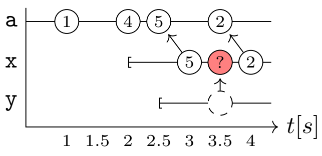
As specified by the `when` expression of the `eval` clause of `x`, it only produces a value when the input `a` is greater than 4. This might not coincide with `a > 3` as required for the evaluation of `y`. Therefore, similar to Example 7, there are points in time when `y` is evaluated, but `x` is not. To detect specifications with such errors, RTLola uses another type system reasoning about the when conditions, called semantic types in that context. Like the pacing type system, the semantic type system ensures the implication relation between semantic types holds. In the above example, the type system would reason whether the implication $a > 3 \rightarrow a > 4$ is a tautology and consequently reject this specification.

Besides ensuring that the *when* conditions of the eval clauses of dependent streams imply each other, the semantic type system also reasons about the lifecycle of dependent streams. Concretely, for a synchronous access to succeed, the accessed stream must be alive at least as long as the accessing stream.

Semantic Types and Periodic Streams. While the previous intuition holds for event-based streams, synchronous accesses between periodic streams imply stricter requirements. Consider the following example:

Example 11 (Shifted Periodicity).

```
input a: Int
output x
    spawn when a > 3
    eval @1Hz with a.hold(or: 0)
output y
    spawn when a > 4
    eval @1Hz with x
```



Here the streams x and y have the same frequency and the spawn condition of y implies the spawn condition of x. However, the synchronous access might fail if x spawns before y since the frequencies are now out of sync. For example, the sequence of events depicted on the right of the example will lead to an invalid synchronous access. First, a has the value 1 and since no spawn condition is true both streams are not spawned. Next, a gets the value 4 spawning x but not y. The later stream is spawned half a second later, which results in the two periods of x and y not being synchronized. This leads to the failure of the synchronous access. To circumvent this problem, the semantic type system requires equality instead of the implication of the spawn and close condition of two dependent periodic streams.

Type System Decidability. Pacing types are defined as positive boolean formulas over input stream names or as fixed frequencies. Hence, it is efficiently decidable if their implication is a tautology. On the contrary, semantic types are arbitrary stream expressions. As described by Schwenger [21], whether an implication between stream expressions is a tautology is generally undecidable. Nevertheless, the RTLola frontend provides a sound over-approximating implementation of the semantic type checker based on syntactic equality. Here, semantic types are parsed as a boolean formula, so implications such as `a.hold(or: 0)` $\wedge$ `b` $\rightarrow$ `a.hold(or: 0)` can still be proven.

6 Parameterization

The specifications in the previous sections were limited to a single intruder. However, in reality, the monitor must observe an unbounded amount of intruders since we can not give an apriori bound on their number. This monitor will inevitably require unbounded memory, yet keeping the memory footprint of the monitor predictable is essential for CPS. For this, RTLola features parameterized output streams [12] that provide a declarative and predictable way of handling unbounded memory through stream expressions.

Parameterized streams lift output streams from a single instance to a set of stream instances. While all stream instances share the stream expressions, each instance of a stream has a different assignment of parameter values. This assignment is determined by an additional stream expression preceded by the `with` keyword in the `spawn` clause. If an output stream is parameterized over multiple parameters, this expression returns a tuple of values matched position-wise to the parameters. Parameters are declared as a comma-separated sequence in braces after the stream name. The spawn clause of a parameterized stream determines when an instance is created as introduced in Sect. 5. The evaluation and close clauses of parameterized streams are evaluated for each instance, determining their value and lifecycle. As for dynamic streams, a stream instance will not be spawned again if an instance with the parameter values already exists.

Consider the final iteration of the running example which extends each output stream with a parameter for the different intruders:

Example 12 (Multiple Intruder).

```
import math
input lat: Float
input lon: Float
input intruder_id: UInt
input intruder_lat: Float
input intruder_lon: Float

output intruder_pos(id)
   spawn with intruder_id
   eval when id = intruder_id with (intruder_lat, intruder_lon)
   close @true when stale(id).hold(or: false)
output distance(id)
   spawn with intruder_id
   eval @((intruder_id && intruder_lat && intruder_lon) || (lat &&lon))
   with sqrt((intruder_pos(id).hold().0.defaults(to: 0.0) - lat.hold(or: 0.0))**2.0 +
        (intruder_pos(id).hold().1.defaults(to: 0.0) - lon.hold(or: 0.0))**2.0)
   close @true when stale(id).hold(or: false)
output closer(id)
   spawn with intruder_id
   eval with distance(id).offset(by: -1).defaults(to: distance(id)) >= distance(id)
   close @true when stale(id).hold(or: false)
output stale(id)
   spawn with intruder_id
   eval @10s with intruder_pos(id).aggregate(over: 10s, using: count) = 0
   close @true when stale(id).hold(or: false)

trigger(id)
   spawn when distance(intruder_id).hold(or: 1.0) < 0.1 with intruder_id
   eval @1Hz when closer(id).aggregate(over_exactly: 5s, using: forall).defaults(to:
        false) with "Intruder {{}} detected".format(id)
   close @true when stale(id).hold(or: false)
```

Notice the additional input stream `intruder_id`. We assume that every intruder has a unique ID provided to the monitor with every update of the `intruder_lat` and `intruder_lon` streams. The output stream `intruder_pos` is added to accumulate these positions on a per-intruder basis. It has a single parameter representing the intruder ID. This is made explicit through its `spawn with` expression that synchronously accesses the `intruder_id` stream. As a result, a new instance of this stream is created for each fresh intruder when it is received for the first time. All other output streams share the same parameterization, so each output stream has an instance for each non-stale intruder ID. The trigger features the additional `spawn when` condition as before. Finally, this specification achieves the goal outlined in Sect. 2 and handles multiple moving intruders efficiently and predictably.

For Experts: Instance Aggregations

Like sliding window aggregations, RTLola features instance aggregations to aggregate the most recent values of the instances of a parameterized output stream. For example, the trigger from Example 12 can be rewritten without parameters using an instance aggregation *as follows*:

```
output approaching(id)
    spawn when distance(intruder_id).hold(or: 1.0) < 0.1 with intruder_id
    eval @1Hz with closer(id).aggregate(over_exactly: 5s, using:
         forall).defaults(to: false)
    close @true when stale(id).hold(or: false)

trigger @true approaching.aggregate(over_instances: all, using: exists) "Intruder
      detected"
```

The `approaching` stream is similar to the trigger from Example 12. The non-parameterized trigger now aggregates over all instances of this new stream using a disjunction. Semantically, this means that whenever any instance of the helper stream evaluates to true, the instance aggregation in the trigger will also evaluate to true, causing the trigger to activate.
Instead of aggregating all stream instances, it is sometimes desirable to only aggregate the instances that produced a *fresh* value in this evaluation cycle. This is specified by stating `over_instances: fresh` in the aggregation. This will couple the pacing of the caller of the aggregation to the evaluation pacing of the target stream of the aggregation. Other instance aggregation functions in RTLola are `forall`, `avg`, `sum`, `count`, `min`, and `max`.

7 Development and Integration

During the previous sections, the examples featured links to the RTLola playground, a web-based implementation of the RTLola frontend and interpreter that can analyse and execute specifications locally in your browser without requiring installation. A version of this tutorial is also available there[1], such

[1] https://rtlola.org/playground/tutorial.

that the specifications and examples from this tutorial can easily be experimented with by pressing the *Copy to Editor* button below them and clicking *Run*. The specifications are tested against a trace simulated using the *Microsoft Flight Simulator*. A video of this trace is included in the overview chapter in the playground, and its raw data can be inspected by switching to the *Trace* tab on the right.

Nonetheless, in real applications, it is necessary to run the monitor either natively or incorporate them within existing applications. The RTLola interpreter provides solutions for both: A library to seamlessly integrate the monitor into Rust applications, along with a standalone command-line application.

7.1 *RTLola* CLI

The simplest way to run the RTLola interpreter locally is the command-line application `rtlola-cli`. The installation of the application can be achieved using the cargo package manager:

```
cargo install rtlola-cli
```

The application offers two modes of execution:

- *Analyze* In this mode, the RTLola frontend is employed to check the provided specification for correctness. This involves checking for syntax errors, ensuring well-definedness and detecting type-related errors.
- *Monitor* This mode enables the execution of a monitor based on the provided specification. After checking the specification for correctness, the monitor can be run in an offline or online setting. In an offline setting, the monitor analyzes a prerecorded trace, while in an online setting, the data arrives in real-time.

In the offline setting, the interpreter monitors a prerecorded trace in CSV format such as the following:

```
a,b,time
1,2,0
#,3,1
3,#,2
```

This example trace defines three events occurring at times 0 s, 1 s, and 2 s, respectively, and assigns new values to two input streams called `a` and `b`. Notably, the hashmark `#` signifies that the corresponding input stream does not receive a new value at that particular time. Subsequently, we can run the `rtlola-cli` to monitor the specification `specification.lola` on this trace:

```
rtlola-cli monitor \
    --offline relative \
    --csv-in trace.csv \
    specification.lola
```

Here, the argument `--offline relative` specifies the type of time format utilized in the "time" column of the CSV file. In the example, the timestamp is given as time in seconds relative to the start of the monitor. Upon executing, each event in the CSV file is forwarded to the monitor, with the resulting output printed to the standard output.

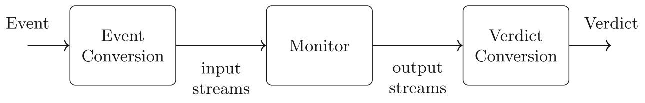


Fig. 5. Process to integrate the monitor

In the online mode, the interpreter retrieves the inputs from a buffer and utilizes the real-time timestamps of the events when they arrive. The following command starts the monitor in an online setting:

```
rtlola-cli monitor --online --stdin specification.lola
```

Here, the application waits for new events on standard input and forwards them to the monitor as soon as they arrive.

We illustrated two instances of using the RTLola interpreter command line application. For a comprehensive list of available command-line arguments and options, you can consult the documentation by executing the following command:

```
rtlola-cli monitor --help
```

The RTLola interpreter has successfully been employed to monitor drones in cooperation with the German Aerospace Center (DLR) and the aircraft manufacturer Volocopter. We identified a set of different monitoring applications, as reported by Baumeister et al. [2], which we discuss in the expert section.

For Experts: Monitoring Applications

The monitor can provide valuable feedback during the development of new components in the aerospace domain. In this safety-critical domain, predefined standards ensure that the concept of operation, requirements, design, and implementation are coherent and include several validation steps. We identified different applications in which monitoring can be used during the development of new components following such standards but also during the operation of these components:

1. *Debugging* The monitor provides feedback to the developer of the component. During the execution, the monitor checks whether the component works as intended and collects statistical information. The developer writes the specification and has access to the internal state of the component.
2. *Validation* The monitor is used to validate the component externally. The specification is written independently and has only access to the inputs and outputs of the component and not to the internal state. This application is, among others, helpful in validating that components by external companies follow their requirements and can be trusted.
3. *Pre-Post-Flight Analysis* This application uses monitoring to check whether all necessary components are operational. The monitor runs predefined test cases and validates that no irregular behavior is detected.

After the flight, the monitor computes more sophisticated information for better evaluation of the flight or runs new specifications on past flights.
4. *In-Flight Analysis/Safe Integration* The monitor provides feedback about the safety of the drone during its operation. It validates the correctness of individual components to ensure a safe flight and reports to the pilot if a property is violated.

All these applications require the integration of the monitor into the development process. Previously, we utilized the command-line application of the RTLola interpreter to execute the monitor. In the following example, we demonstrate how to integrate the interpreter using the Rust library.

Integration with the RTLola API. We assume that the monitor is running on the drone and receives the input data over internal communication. The output of the monitor is then sent over TCP to a ground station displaying the trigger messages of the monitor so a pilot can take over. Figure 5 illustrates two steps for this integration process. As shown in the figure, the setup consists of three components: The monitor, in the center of the figure, is automatically generated from the specification and does not require further integration. However, two interfaces must be implemented to handle the communication with the system and the operator. These implementations are specific to the setup: The *Event Conversion* receives the incoming sensor readings and transforms the data into an internal representation the monitor understands. The *Verdict Conversion* transforms the internal representation of the monitor's output to messages accepted by the ground station.
After providing all the missing implementations, we need to configure the monitor and start the evaluation. This results in the following code, skipping the concrete configuration of the `event_source` and the `verdict_sink`:

```
let mut monitor = ConfigBuilder::new()
    .with_spec(spec)
    .online()
    .with_event_factory::<ExampleInputs::Factory>()
    .with_verdict::<TriggerMessages>()
    .monitor()?;
let mut event_source = ...
let mut verdict_sink = ...
while let Some((ev, ts)) = event_source.next_event()? {
    let verdicts = monitor.accept_event(ev, ts)?;
    verdicts_sink.sink_verdicts(&verdicts)?;
}
```

Event Conversion. The Event Conversion receives sensor values and transforms the readings into an internal representation the monitor uses. In our setup, we receive the sensor values over a UDP connection as a byte-stream and differentiate between two types of messages: The first type of message is sent by the GNSS sensor that computes the latitude and longitude of the drone. The second type of message contains the latitude and longitude of the intruders. The byte-stream needs to be parsed and converted to map the incoming data to the corresponding input streams. The implementation in our setup uses the simplified interfaces shown in Fig. 6. After providing a parser function, the implementation receives the byte stream and parses this stream to an event called `ExampleInputs`. The implementation of the interface is provided automatically by the macros `ValueFactory` and `CompositeFactory`. These macros generate code for a factory that maps, for example, the `lat` field in the `Gnss` struct to the `lat` input stream in the monitor.

Verdict Conversion. The Verdict Conversion transforms the internal representation of the monitor's output into messages in a form expected by the ground station. In our setup, this conversion interprets trigger messages as bytes and sends these over TCP to the ground station. The interfaces for this setup are implemented generically, so no further steps are needed.

```
impl ByteParser for DroneExampleParser {
    fn from_bytes(&mut self, bytes: &[u8]) -> Result<(ExampleInputs, usize)> { ... }
}
```

```
#[derive(ValueFactory)]
struct Gnss {
    lat: Float64,
    lon: Float64
}
```

```
#[derive(ValueFactory)]
#[factory(prefix)]
struct Intruder {
    id: UInt64
    lat: Float64,
    lon: Float64
}
```

```
#[derive(CompositFactory)]
enum ExampleInputs {
    Gnss(Gnss),
    Intruder(Intruder)
}
```

Fig. 6. Concrete Implementation of the Event Conversion

8 Conclusion

The running example from our tutorial has illustrated the expressiveness of the RTLola specification language, which makes RTLola well-suited for complex application domains like aerospace. The development of the specifications is facilitated by the comprehensive support of the tool framework. Since the same specification can be used in multiple different settings, a specification can be validated early in a test environment or on log data, long before the monitor is integrated into the aircraft; the automatic analysis of the specification furthermore ensures that the monitor operates correctly and reliably.

RTLola has been very successful in the aerospace domain (cf. [2,4]). RTLola has also been used in other cyber-physical applications, including cars [6] and medical equipment [14], and in domains beyond CPS, such as networks [12]. The combination of the highly expressive RTLola specification language with the reliability obtained by static analysis and the resource efficiency of the monitoring framework is of great use in all these settings.

Data Availability Statement. The artifacts and resources associated with this paper are accessible as follows:
1. Primary Artifact: The primary artifact of this paper is available via: https://doi.org/10.5281/zenodo.12633784.
2. Source Code: The source code of the framework is hosted on GitHub:
RTLola Interpreter: https://github.com/reactive-systems/RTLola-Interpreter
RTLola Frontend: https://github.com/reactive-systems/RTLola-Frontend
3. Software Packages: The relevant software packages are available on crates.io:
RTLola Interpreter: https://crates.io/crates/rtlola-interpreter
RTLola Frontend: https://crates.io/crates/rtlola-frontend

References

1. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
2. Baumeister, J., et al.: Monitoring unmanned aircraft: specification, integration, and lessons-learned. In: Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, Canada, 22–27 July 2024. Accepted for publication (2024)
3. Baumeister, J., Finkbeiner, B., Kruse, M., Schwenger, M.: Automatic optimizations for stream-based monitoring languages. In: Deshmukh, J., Ničković, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 451–461. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_25
4. Baumeister, J., Finkbeiner, B., Schirmer, S., Schwenger, M., Torens, C.: RTLola cleared for take-off: monitoring autonomous aircraft. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 28–39. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_3
5. Baumeister, J., Finkbeiner, B., Schwenger, M., Torfah, H.: FPGA stream-monitoring of real-time properties. ACM Trans. Embed. Comput. Syst. **18**(5s), 88:1–88:24 (2019). https://doi.org/10.1145/3358220
6. Biewer, S., Finkbeiner, B., Hermanns, H., Köhl, M.A., Schnitzer, Y., Schwenger, M.: On the road with RTLola. Int. J. Softw. Tools Technol. Transf. **25**(2), 205–218 (2023). https://doi.org/10.1007/S10009-022-00689-5
7. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: temporal stream-based specification language. In: Massoni, T., Mousavi, M.R. (eds.) SBMF 2018. LNCS, vol. 11254, pp. 144–162. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03044-5_10
8. D'Angelo, B., et al.: LOLA: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23–25 June 2005, Burlington, Vermont, USA, pp. 166–174. IEEE Computer Society (2005). https://doi.org/10.1109/TIME.2005.26
9. Dauer, J.C., Finkbeiner, B., Schirmer, S.: Monitoring with verified guarantees. In: Feng, L., Fisman, D. (eds.) RV 2021. LNCS, vol. 12974, pp. 62–80. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88494-9_4
10. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press (2013). https://doi.org/10.3233/978-1-61499-207-3-141

11. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. Int. J. Softw. Tools Technol. Transf. **23**(2), 255–284 (2021). https://doi.org/10.1007/S10009-021-00609-Z
12. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 152–168. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_10
13. Faymonville, P., et al.: StreamLAB: stream-based monitoring of cyber-physical systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 421–431. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_24
14. Finkbeiner, B., Keller, A., Schmidt, J., Schwenger, M.: Robust monitoring for medical cyber-physical systems. In: Proceedings of the Workshop on Medical Cyber Physical Systems and Internet of Medical Things, MCPS 2021, pp. 17–22. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3446913.3460318
15. Finkbeiner, B., Kohn, F., Scheerer, F., Schledjewski, M.: The RTLola Playground (2023). https://rtlola.org/playground
16. Finkbeiner, B., Kohn, F., Schledjewski, M.: Leveraging static analysis: an IDE for RTLola. In: André, É., Sun, J. (eds.) ATVA 2023. LNCS, vol. 14216, pp. 251–262. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-45332-8_13
17. Finkbeiner, B., Oswald, S., Passing, N., Schwenger, M.: Verified rust monitors for lola specifications. In: Deshmukh, J., Ničković, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 431–450. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_24
18. Gorostiaga, F., Sánchez, C.: Striver: stream runtime verification for real-time event-streams. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 282–298. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_16
19. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebraic Methods Program. **78**(5), 293–303 (2009). https://doi.org/10.1016/J.JLAP.2008.08.004
20. Schwenger, M.: Monitoring cyber-physical systems: from design to integration. In: Deshmukh, J., Ničković, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 87–106. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_5
21. Schwenger, M.: Statically-analyzed stream monitoring for cyber-physical systems (2022). https://doi.org/10.22028/D291-37014
22. Torfah, H.: Stream-based monitors for real-time properties. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 91–110. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_6

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Author Index

© The Editor(s) (if applicable) and The Author(s) 2025
A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 649–652, 2025.
https://doi.org/10.1007/978-3-031-71177-0