Universität der Bundeswehr München

Fakultät für Informatik
Institut für Technische Informatik

A Specification Language for Reconfigurable Dependable Systems, its Formalisation and Analysis Environment

Martin Riedl

Vollständiger Abdruck der von der Fakultät für Informatik der Universität der Bundeswehr München zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Promotionsausschuss

Vorsitzender:	Prof. Dr.-Ing. Andreas Karcher [1]
1. Berichterstatter:	Prof. Dr.-Ing. Markus Siegle [1]
2. Berichterstatter:	Prof. Dr.-Ing. Samuel Kounev [2]
1. Prüfer:	Prof. Dr.-Ing. Mark Minas [1]
2. Prüfer:	Prof. Dr. rer. nat. Stefan Pickl [1]
3. Prüfer:	Prof. Klaus Buchenrieder, Ph.D. [1]

[1] Universität der Bundeswehr München
[2] Universität Würzburg

Die Dissertation wurde am 17.3.2014 bei der Universität der Bundeswehr München eingereicht und durch die Fakultät für Informatik am 14.7.2014 angenommen. Die mündliche Prüfung fand am 15.7.2014 statt.

Abstract

Model-based dependability evaluation is challenging and therefore prone to error, especially if one uses lower-level formalisms such as Markov Chains, Stochastic Petri Nets or Stochastic Process Algebra (SPA). For that reason, in an industrial environment mainly high-level specification formalisms such as AADL, UML or SysML are employed. Those languages mostly lack formal semantics and capabilities to model stochasticity which is an intrinsic characteristic of dependable systems.

For the purpose of bridging the gap between high-level formalisms and formal modelling languages, this thesis formalises the *LAnguage for REconfigurable dependable Systems* (LARES) regarding its syntax and semantics. It can serve both as an intermediate language and as a stand-alone modelling formalism. LARES provides language elements for hierarchical modelling. It separates between structure and behaviour. Furthermore, it introduces scopes which restrict the visibility of definitions and named statements in order to facilitate structured model descriptions. The semantics of LARES is given by means of SPA and labelled transition systems (LTS). For this purpose, two transformations have been realised which can fully automatically map a LARES model to either its SPA equivalent or directly to an LTS. A number of implementations are described which constitute the LARES modelling environment: The editor component supports syntax highlighting and code completion. It also checks context conditions in order to ensure model validity. The analysis component carries out the transformations of a given model and allows visualising the constructed instance tree, the composed state space or the calculated dependability measures. And finally, among other things, the library implements language definitions, parsers, transformations and bindings to external solvers. Both transformations can be verified as the resulting SPA model and the LTS have to correspond regarding their behaviour. In addition, a proof sketch is provided to show the equivalence of both transformation semantics. Several example models and a real-world model (the active suspension system of an autonomous RailCab vehicle) are presented which were modelled and analysed in order to confirm a sufficient expressiveness and to collect metrics of constructed LARES specifications, arising SPA models and symbolic representations during analysis. These metrics serve as an indicator to reason about scalability regarding the language constructs which have been used to capture a model's combinatorial complexity. The thesis also compares LARES to several related approaches and concludes with numerous ideas on possible future extensions and improvements.

Zusammenfassung

Modellbasierte Verlässlichkeitsbewertungen sind anspruchsvoll und daher fehleranfällig, insbesondere wenn Formalismen niedriger Abstraktionsstufe wie Markovketten, Stochastische Petri-Netze oder Stochastische Prozessalgebren (SPA) genutzt werden. In der industriellen Anwendung finden daher häufig Spezifikationssprachen mit höherem Abstraktionsgrad wie AADL, UML oder SysML Verwendung. Diese sind jedoch zumeist weder formal definiert noch ermöglichen sie die Modellierung zufälliger Ereignisse, ein intrinsisches Merkmal verlässlicher Systeme.

Die vorliegende Dissertation greift dieses Problem auf. Sie formalisiert die Syntax und Semantik der *LAnguage for REconfigurable dependable Systems* (LARES), die sich sowohl als eigenständige Modellierungssprache als auch als Zwischendarstellung nutzen lässt. LARES bietet Sprachmittel zur Modellierung von Hierarchien. Hierbei unterscheidet LARES zwischen Struktur und Verhalten und trägt durch Sichtbarkeitsbereiche (von Definitionen und referenzierbaren Ausdrücken) zur Strukturierung von Modellbeschreibungen bei. Die Semantik der Sprache wird mit Hilfe von SPA und beschrifteten Transitionssystemen (LTS) definiert. Dazu werden zwei Transformationen realisiert, die ein LARES Modell automatisch in dessen SPA Darstellung oder direkt in ein LTS überführen. Zudem erläutert diese Dissertationsschrift alle Implementierungen, die zusammen die LARES Modellierungsumgebung bilden: Die Editorkomponente hebt die Syntax visuell hervor, vervollständigt automatisch Ausdrücke und prüft Kontextbedingungen zur Gewährleistung der Modellvalidität. Die Analysekomponente transformiert die Modelle und visualisiert den erzeugten Instanzbaum, den Zustandsraum oder die berechneten Verlässlichkeitsmaße. Die Bibliothekskomponente beinhaltet unter anderem Sprachdefinitionen, Parser, Transformationen und Anbindungen an externe Analysewerkzeuge. Beide Transformationen können auf ihre Korrektheit geprüft werden, da sich die erzeugten SPA- und LTS-Modelle hinsichtlich ihres Verhaltens entsprechen müssen. Des Weiteren wird ein Beweis skizziert, der die Äquivalenz dieser Transformationen zeigt. Mehrere Anwendungsbeispiele, unter anderem ein aktives Dämpfungssystem eines autonomen RailCab-Fahrzeugs, bestätigen die ausreichende Ausdrucksmächtigkeit. Sie dienen auch dazu, Metriken über die jeweilige LARES Spezifikation, die temporäre SPA Darstellung sowie die symbolische Kodierung während der Analyse zu bestimmen. Die Metriken geben Hinweis darauf, wie der LARES-Ansatz skaliert, abhängig von den genutzten Sprachkonstrukten zur Abbildung der kombinatorischen Komplexität der Modelle. Die Dissertation vergleicht und grenzt LARES gegenüber verwandten Ansätzen ab und schließt mit einigen Ideen über zukünftige Erweiterungs- und Verbesserungsmöglichkeiten.

Contents

Chapter 1

Introduction

The correct and timely functioning of increasingly complex systems, e.g. in the communication, transportation or energy sectors is crucial. Therefore, we aim to understand and to improve the dependability of these systems. When dealing with fault-tolerant systems, system design and reconfiguration are sensitive aspects concerning a system's non-functional behaviour such as reliability, availability, survivability and others. To ensure that a certain requirement concerning these non-functional behaviours is met, several methods and tools for modelling and quantitative evaluation have been developed. Exhaustive analysis or rare-event simulations are established methods to be applied, as cases of failures are 'hopefully'extremely rare by nature [38]. Especially tools which apply exhaustive methods mostly come from academia. There, the research is mainly focused on expressiveness and theoretic verifiability of models specified with the help of formal languages, or speeding up the analysis methods. Indeed, formalisms such as reliability block diagrams (RBD) or fault trees (FT) are used by engineers to design fault tolerant systems [52], but suffer from expressiveness (e.g. regarding temporal characteristics and mutual dependencies between components). State-based formalisms such as stochastic Petri nets (SPN) [112] or Markov chains (MC) [20, 105] do not pervade common usage. Especially Markov models put a lot of burden on modellers to consider all reachable combinations of component states. Compositional languages such as stochastic process algebra (SPA) [45] seem to fill this gap, but still do not attract sufficiently – probably because it is a non-trivial task to specify process synchronisation correctly. Thus, despite the compositionality of stochastic processes, it remains difficult to express component interaction by the available language means.

As a consequence many modellers do not use these concepts, since it turned out to be a hurdle and too complicated for non-specialists. This work is subjected to the assumption that modellers want to stay in their well-known formalisms or else to adopt a language which is as similar as possible to the formalisms and tools they are used to. Some important aspects have been identified which imply a number of crucial requirements for a fault-tolerant modelling language, which have also been considered by others (cf. [27] etc.). These include

- language means for mapping the (hierarchical) structure of a system,
- interface definitions as a basis for modularisation,
- simple interaction specifications at each level within the hierarchy,
- the ability to parametrise a model,
- atomic processes to represent stochastic behaviour at least in terms of discrete probabilities, or better, continuous probability distributions and, most important,
- high expressiveness.

Defining failure combinatorics as simply as with FTs and RBDs, whilst dealing with non-Boolean and dynamic behaviour has not been achieved by any other approach. The explanation and the analysis of the considered approaches are detailed in Chapter 6. There, pros and cons which finally encouraged us to define a language to effectively encounter the posed requirements will be examined. This language is named LARES — Specification LAnguage for REconfigurable Systems — and hence emphasising explicitly that a fault-tolerant-system may also dynamically adapt to specific situations by reconfiguration. Its name comes from ancient times, when knowledge was still poor and many things remained obscure and mysterious, for instance the Romans believed in guardian deities to care for their safety:

> “Lares were believed to observe, protect and influence all that happened within the boundaries of their location or function. [...] Lares are sometimes categorised as household gods but some had much broader domains. Roadways, seaways, agriculture, livestock, towns, cities, the state and its military were all under the protection of their particular Lar or Lares.” [140].

Over time societies established mechanisms and institutions which help protecting individuals as well as society as a whole. We still rely on these achievements. To obtain further knowledge on how things interact and to comprehend which event causes a certain effect are important contributions. Modern societies commonly do not believe in the supernatural as an explanation for the inexplicable. Instead, they rely on scientific results, technology and systems they build to directly influence their environment, to support and to protect them. In response to these needs, we developed a framework which implements the LARES language and provides tools for modelling, editing, annotation, transformation and analysis of given models. LARES allows defining complex error behaviour, non-Boolean components, systems with independent components and systems with intrinsic non-monotonicity.

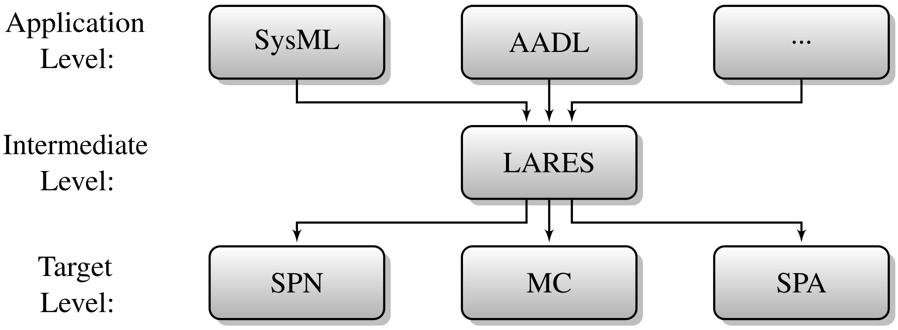


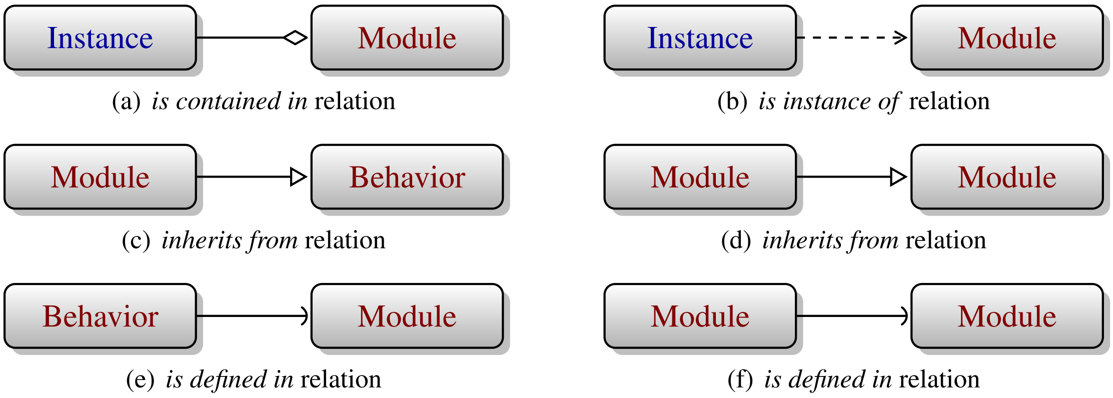
Figure 1.1: Hourglass transformation of application level specifications into formal target level models using LARES as intermediate representation

The aim is to keep maximum expressiveness whilst maintaining usability and modularity. Modellers (e.g. an engineer) nowadays are commonly trained on object-oriented programming (OOP) languages or at least on structural languages. The OOP-style language definition is assumed to feel familiar to them. Hereby, automated transformations ensure the elimination of additional sources of error (that could occur when translating manually to a formal target language). The formalism should serve as both a stand-alone input language and an intermediate language for a temporary representation when transforming from an application-level formalism such as AADL (Architecture Analysis & Design Language [130]), SysML (Systems Modeling Language [109]) and others into a number of formal target languages such as an SPA or, finally, an MC as depicted in Figure 1.1.

In order to give the reader an overview of LARES, the following sections will provide an informal definition of the language features and their meaning, illustrated subsequently by the use of two distinct example models. Eventually the contribution of LARES to the domain of performance and dependability modelling will be elucidated as well as the organisation of this thesis.

1.1 Informal Introduction to LARES

In order to become acquainted with LARES as a dependability modelling formalism, this section provides an overview of the linguistic means of LARES for structuring a model, for specifying behaviours, interactions and measures. The latter is used in order to calculate the probabilities associated with certain sets of states of the model. Note that the syntax sketched in this section is simplified and does not reflect the full syntax in order to suffice only the bare necessities used to denote the running examples. The full syntax is specified in Chapter 2.



(a) *is contained in* relation

(b) *is instance of* relation

(c) *inherits from* relation

(d) *inherits from* relation

(e) *is defined in* relation

(f) *is defined in* relation

Figure 1.2: Structural associations available in LARES in an UML-like notation

Hierarchical Structure: Abstract Definitions

At the root of a LARES specification, a number of definitions can be stated. Those definitions comprise three kinds of types, i.e. Module definitions, Behavior definitions and a single System definition. A Behavior defines the *sequential behaviour*, i.e. an automaton which has states and transitions. Inside a Module definition the interaction among Behavior subinstances is described (interaction behaviour). The advantage of splitting sequential behaviour from interaction behaviour is twofold. On the one hand, modelling complexity inside the Behavior definition is decreased in such a way that patterns of interactions are outsourced to superior levels. On the other hand, this distinction can be employed by a graphical editor in order to support specific diagrams for behaviours, interaction and structure. A definition simply represents an abstract type. It therefore has no effect on the overall behaviour of the model as long as it is not instantiated. The scope of a definition encompasses the location of its definition and its associated subtrees. This is similar to the visibility of inner classes in OOP paradigms.

In general, Figure 1.2 shows how these language constructs relate to each other:

- An Instance keyword, which is only allowed to be used inside a Module (cf. Figures 1.2(a)), implies the instantiation of the specified Module type (cf. Figure 1.2(b))
- A Module may inherit from a Behavior or a Module (cf. Figures 1.2(c) and 1.2(d))
- Inside each Module definition further Behavior and Module definitions may be given (cf. Figures 1.2(e) and 1.2(f))

Accordingly, a LARES specification can be structured as illustrated by Figure 1.3 which defines a Module for a network, some system Behavior and the System instance. Since e.g. the Module `MLink` is defined inside the Module definition `MNetwork`, it is not accessible from the System definition `S`.

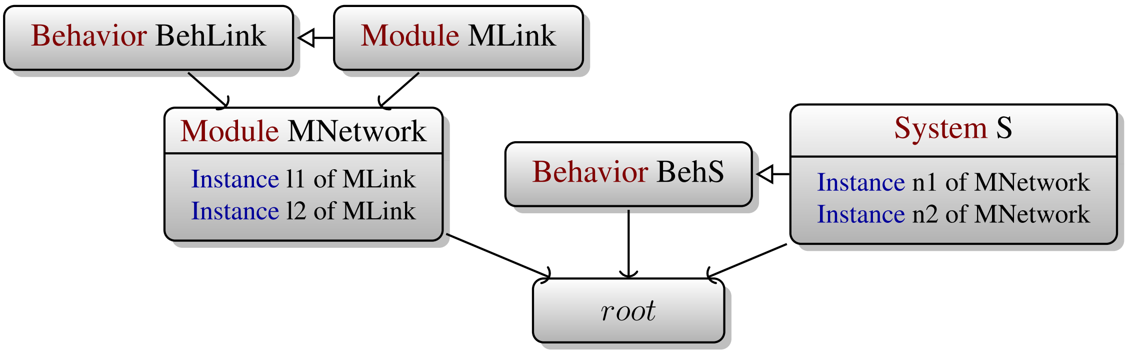


Figure 1.3: Structure of abstract definitions of a LARES example model

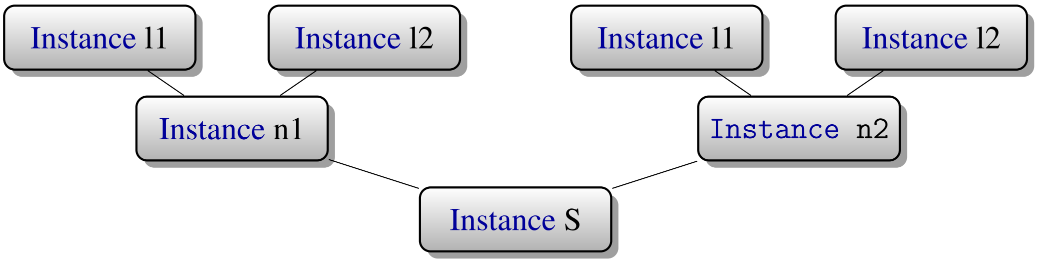


Figure 1.4: Instance tree arising from the LARES example of Figure 1.3

Hierarchical Structure: Instance Tree

The System definition is of twofold meaning: On the one hand, it is a Module definition, whereas on the other hand, it represents an implicit root instance.

As indicated by the example of Figure 1.3, each Module definition may contain further subinstances, e.g. by stating `Instance n1 of MNetwork` the Module definition `MNetwork` is instantiated with the name `n1`. Due to the implicit instantiation of the System definition, a recursive instantiation of all subinstances is triggered. The whole instance tree of the model is hereby constructed. As a Module definition may inherit from an arbitrary number of Behavior or Module definitions, its instantiation recursively instantiates the inherited definitions. The constructed instance tree (cf. Figure 1.4) of the current example given by Figure 1.3 hence conforms the example's definition structure. Note that an instance can also be individually configured by instantiating a parametrised Module.

Instances always have to communicate via parental nodes within the instance tree. According to that, the definition of direct mutual dependencies (interactions) among components of different subtrees is prevented by language definition. As follows, this has some implications on the provided statements (which can be used inside a Behavior or a Module) and their syntax.

Statements within a Behavior Definition

A Behavior defines a sequential process in terms of an automaton. A Behavior definition can be inherited by a Module definition, whereas Behavior definitions cannot inherit from other language elements. The ingredients of Behaviors are the following:

- The State statement explicitly denotes names for a state or a mode. Each name serves as a Boolean variable which is used by its direct environment to decide whether an instance of a Behavior is in a certain state or not.

- The Transitions statement specifies events that imply a state-change (i.e. a *transition*) from a source state to some target state. An event can be guarded by a label, also denoted as guard label, or it can be unguarded. Unguarded transitions are internal and are hence independent from external conditions, whereas guarded transitions can be referenced via their guard labels and triggered by some environmental conditions. A guarded transition can only be executed when its source state is the current state. Transitions may be delayed, by denoting a continuous probability distribution, or immediate, thus following an implicit discrete distribution defined with the help of weights.

A statement that defines a name, which can be locally referenced or by its direct environment, will subsequently be categorised as a *named statement*. State variables or guard labels constitute named statements accordingly.

Statements within a Module Definition

Apart from possibly containing further Module or Behavior definitions, a Module definition may also contain a number of statements:

- An Initial statement is a *named statement* that specifies an initial configuration of the associated instance subtree by means of a vector of references to further Initial statements or State statements. Referring to the example given by Figure 1.3, let `active` and `inactive` be possible Initial configurations of a link. A network which consists of two links `l1` and `l2` can e.g. be initialised by referring to one of the Initial configurations `i1` or `i2`:

  ```
  Initial i1 = l1.active, l2.inactive
  Initial i2 = l1.active, l2.active
  ```

- A Condition statement is a *named statement* whose name represents a Boolean variable which is assigned to a *condition expression*, i.e. a Boolean expression over state variables of subinstances. The atomic elements of the *condition expression* may thereby address a state variable either directly via a Behavior instance or indirectly via further Condition statements (which may be provided locally or by Module subinstances). Let e.g. a link reveal the Boolean variable up specified by a Condition statement. The Boolean variable `working` specifies for a network (of two links `l1` and `l2`) that it works when at least one link is up:

```
Condition working = l1.up | l2.up
```

- A forward statement is a *named statement* which triggers guarded transitions. Its name represents a guard label similar to guarded transitions.

  If multiple guarded transitions need to be addressed by a forward statement, a synchronisation behaviour needs to be defined. This is achieved by means of a *reactive expression* which may contain operators for synchronisation and references to guarded transitions. Here, a sync operator requires all addressed guarded transitions to be able to execute simultaneously. A maxsync operator requires at least one guarded transition to be able to execute and a choose operator requires exactly one of the addressed guarded transitions to be able to execute. Apart from triggering guarded transitions directly, guard labels of other forward statements may be referenced instead.

  In the example below, a repairman `rm` is added to the network with two links, and a repair mechanism is defined in terms of a forward statement. The sync operator allows repairing (via `rm.⟨repair⟩`) only as long as the repairman `rm` is available. The maxsync operator specifies that the links `l1` and `l2` will be immediately repaired (if one link failed, or both at once):

  ```
  forward ⟨repair⟩ to sync{
    rm.⟨repair⟩, maxsync{l1.⟨rep⟩, l2.⟨rep⟩}
  }
  ```

- A guards statement is similar to a forward statement except that the guard label is substituted by a condition expression. Only if the condition expression is satisfied, referenced guard labels can be triggered. The condition expression is therefore also denoted as *generative condition*.

  Note that the introduced terms must not be confused with the notion of generative/reactive/stratified models as used in [64] to classify probabilistic processes. In contrast to LARES, where these terms are used in order to specify which condition yields a specific reaction, generative/reactive/stratified models represent classes of models that inter-relate via abstraction and bisimulation.

As an example, let `n` denote a network which provides a Condition statement `working`. The guards statement is defined within the System definition and triggers a repair mechanism in `n`:

```
!n.working guards n.⟨repair⟩
```

Note that the Condition and the forward statements are necessary, not so much because of avoiding redundant definitions of condition or reactive expressions, but due to implementing the desired strict visibility policies, as it will be described in the next two parts, in order to reason on states or to propagate events across hierarchical boundaries.

Visibility of Statements

As it was already indicated, a named statement is only visible for its direct environment. This means that a named statement can only be accessed locally, where it has been defined, or from the parental node. Accordingly,

- a condition expression can either refer directly to a state of an inherited Behavior instance, or to a Condition statement (defined locally or inside a subinstance of a Module),
- a reactive expression can either refer to guard labels of guarded transitions (inside inherited Behavior instances) or to forward statements (defined locally or inside a subinstance of a Module), and
- an Initial statement can either refer directly to a state of an inherited Behavior instance, or to an Initial statement (defined locally or inside a subinstance of a Module).

The provided named statements constitute an interface. A named statement of an instance can be accessed by referring to the instance name and the name of the statement.

Interaction of Component Instances

An interaction between a number of instances takes place in case that a certain condition is fulfilled and some specified reaction can be performed. Such a condition is specified by referring to Condition or State statements of the desired instances in order to make a claim on these states and, accordingly, to imply a reaction inside the addressed instances by triggering guard labels. For this purpose, a reactive expression is specified which can either directly refer to guard labels of a Behavior instances, or else via forward statements. Figure 1.5 depicts how assertions on states of a subtree can be made using `Condition`

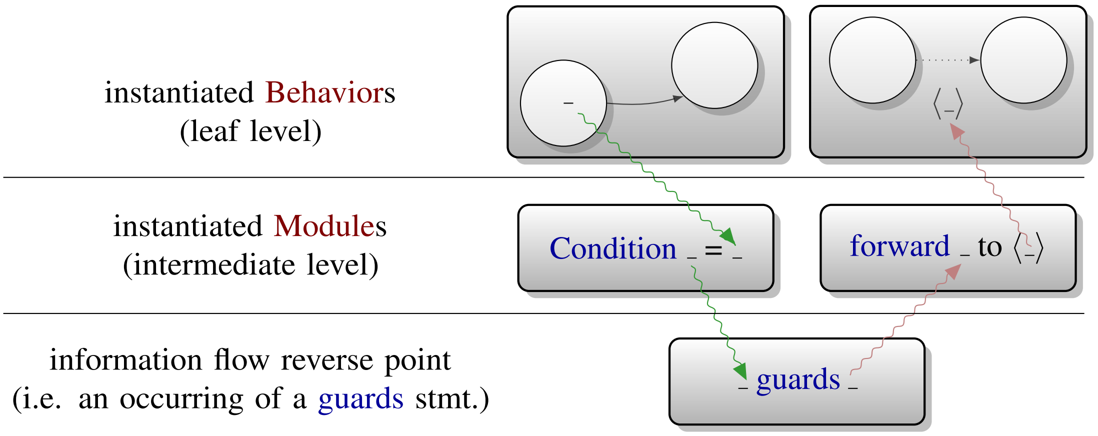


Figure 1.5: Component interaction: Information flow

statements throughout the hierarchy, and how an event is generated by a guards statement in order to trigger a desired reaction, specified by the reactive expression. The underscore _ is a wildcard for a condition expression or a reactive expression. As it was previously explained, a reactive expression may also address more than one instance. Several operators are provided for this purpose, i.e. maxsync, sync and choose, which determine whether a specified reaction can take place and, eventually, how the addressed instances behave in cooperation. All intermediate `forward` statements pass this event through the instance tree until a final Behavior instance will be reached.

Explicit Language Elements for Dependability Modelling

Condition statements are similar to fault trees. They can thus be used to define fault hierarchies throughout the model in order to trigger a certain reaction when used inside a guards statement. There, language means for dependability modelling such as *k-out-of-n* statements can be used by the `oo` keyword. LARES does not provide explicit language elements which describe complex concepts such as Failure-on-Demand, Failure-on-Repair and others. Instead, LARES covers a more general area which also includes performance aspects. In consequence, a modeller has to define the above dependability concepts by himself in terms of Behavior definitions which may be inherited by Module definitions or by specifying interactions among the components of a dependable system.

Specification of Probability Measures for Analysis

Inside a Module definition, the LARES language allows modellers to specify Probability statements in order to address a specific analysis. Two types of analysis methods are currently supported: Transient analysis is performed on the overall state space of the system in order to determine how the probability mass is distributed over the states after a given amount of time, whereas steady-state analysis considers an infinite time horizon in order to determine the stationary probability distribution. A condition expression thereby specifies the states of interest. Their fraction of the probability mass finally represents the result of this measurement.

When e.g. the probability of the network of being at work is desired to be evaluated, the following Probability statements can be specified which trigger a transient analysis (such that the probability at a certain point in time is evaluated) or a steady-state analysis:

```
Probability P_t = Transient(n.working, 10)
Probability P_s = Steadystate(n.working)
```

It lies in the responsibility of the user whether the obtained probability represents, for example, the system reliability, the availability, or other kinds of measures of interest such as survivability. All of them have different definitions which could require users to adapt the model's behaviour, the type of analysis or the Probability statements in order to obtain the measures of interest:

Reliability: The probability of a system to be operational throughout a predefined interval without system-level repair (i.e. becoming operational again)

Availability: The fraction of time in which the system is up

Survivability: The ability of a system to provide a given level of service after the occurrence of a severe event

The next two sections will each provide an example model in order to demonstrate how to apply the language as it was informally introduced in this section and give an informal intuition about its semantics. Both the Fault Tolerant Network and the Component Monitoring System example will serve as vivid examples which can and will be accessed throughout this thesis to exemplify the formal representations shown in the following chapters.

1.2 Fault Tolerant Network Example

A variant of the Fault Tolerant Network example is provided here. It has been presented in its original form in [121]. Imagine a two-processor (p[1] and p[2]) networked system implementing a critical safety function as shown by Figure 1.6. The safety function will be unavailable if either one of the processors or the network fails. The network will fail if all of its links fail. In the provided model the network consists of three links, the first one is initially in use while the others are running in hot standby (i.e. an alternative connection can immediately be established by these redundant links in case of a failure). All the links

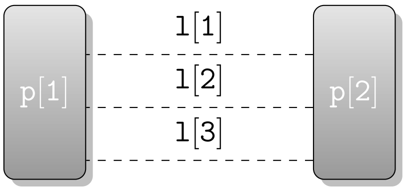


Figure 1.6: Fault tolerant network example with 3 redundant links

(l[1], l[2] and l[3]) can fail. If a link fails, the next passive one becomes active. When the first link fails, a repair unit is notified starting the repair process which repairs all failed links at once and switches them to their initial configurations.

In Figure 1.7 a notation similar to UML is used to depict a possible structure of the model. The structure serves as a template to specify the model using LARES. In the following paragraphs parts of the model will be defined in valid syntax: At first, the repair behaviour is defined, since this behaviour is associated with the network. Then, the network itself is defined which includes the Behavior definition of the link, the Module definition of the link, the network's subinstantiations and the interactions among them. And finally, a Condition is defined stating whether the network failed or not.

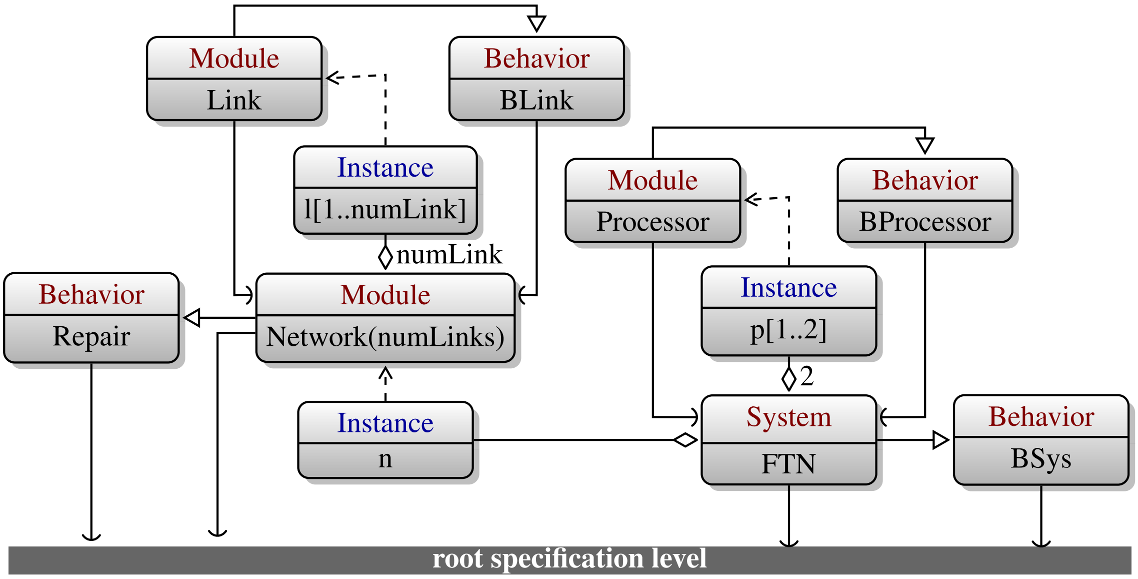
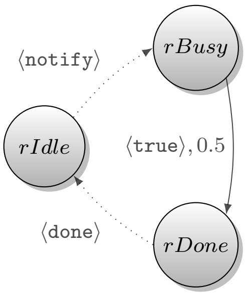


Figure 1.7: FTN: The model structure

The repair behaviour is textually specified as follows and includes a graphical representation in which dotted lines correspond to guarded transitions with unspecified distributions (that are implicitly determined by the interacting partners, but can be regarded as discrete for this example):

```
Behavior Repair {
   State rIdle, rBusy, rDone
   Transitions from rIdle if ⟨notify⟩ → rBusy
   Transitions from rBusy
    → rDone, delay exponential 0.5
   Transitions from rDone if ⟨done⟩ → rIdle
}
```



It consists of three states named `rIdle`, `rBusy` and `rDone`. When the repair behaviour is in `rIdle`, the signal `notify` causes the behaviour to start the repair process. After an exponentially distributed delay `rDone` is reached. An interaction may be triggered via `done` by which the failed links get repaired and the repair unit reach the `rIdle` mode.

The next step is to define the network:

```
Module Network(numLink) : R←Repair { ... }
```

The number of redundant links that a network should provide is left open by introducing a parameter `numLink` for this model. The notation `R<-Repair` adds a repair behaviour (referable via the name `R`) to the `Network`.

The following paragraphs will provide the content of the body of the `Network` definition. As it can be studied from Figure 1.7, two definitions are contained by the body. The Behavior definition `BLink` is listed first:

```
Behavior BLink {
   State lstandby, lactive, lfailed
   Transitions from lstandby
     if ⟨swactive⟩ → lactive
     if ⟨true⟩ → lfailed, delay exponential 0.1
   Transitions from lactive
     if ⟨true⟩ → lfailed, delay exponential 0.3
     if ⟨swstandby⟩ → lstandby
   Transitions from lfailed
     if ⟨swactive⟩ → lactive
     if ⟨swstandby⟩ → lstandby
}
```

A link can fail in both states `lstandby` and `lactive` after a certain delay caused by different exponential failure rates of the transitions to the state `lfailed`. An expression of the form `if <true>` hereby specifies explicitly that a transition is not guarded. The expression `if <true>` can alternatively be omitted. Since a link must be able to switch between its functional modes, it offers guarded transitions for activation to the state `lactive` or deactivation to the state `lstandby`. Furthermore, two guarded transitions are provided in the case of being repaired: The link is either repaired and set to its activated mode or else to its standby mode.

The Module definition `Link` provides a number of named statements constituting an interface: Two conditions, i.e. `cFailed` and `cStandby`, the forward labels `<swactive>` and `<swstandby>` and two initial operational modes, i.e. `iActive` and `iStandby`:

```
Module Link : BLink {
  Condition cFailed = BLink.lfailed
  Condition cStandby = BLink.lstandby
  forward ⟨swactive⟩ to BLink.⟨swactive⟩
  forward ⟨swstandby⟩ to BLink.⟨swstandby⟩
  Initial iActive = BLink.lactive
  Initial iStandby = BLink.lstandby
}
```

All of these statements relate parts of the interface of the inherited behaviour `BLink` one-to-one to the interface of the link's Module definition. The state `lactive` of `BLink` remains hidden to the environmental components, since it is not explicitly addressed by a condition. By definition, the initial state of `BLink` is `lstandby` (as it is the first occurring state). The Module definition `Link` additionally allows addressing `lactive` as the initial state via `iActive`.

Having defined the Module definition `Link`, the network's subinstantiations can be specified. Since the Module definition `Network` consist of `numLink` `Link` instances `l[1]` to `l[numLink]`, where `l[1]` is initialised in active mode in contrast to the others that are initialised in standby mode. For a number of cases guards statements have to be defined representing rules to switch the next possible link `i` to active if link `1` to `i-1` have failed. Note that `i` is already fixed for that case:

```
(AND[j in {1 .. i−1}] l[j].cFailed) & l[i].cStandby
  guards l[i].⟨swactive⟩
```

The Condition statement `nfailed` allows the environment to assert whether the network has failed by violating the redundancy structure:

```
Condition nfailed = AND[i in {1 .. numLink}] l[i].cFailed
```

Furthermore, a repair mechanism is specified. A guards statement has to notify the repair behaviour instance once the first link will have failed. A second one states that if the repair is performed, the links will have to switch back to their initial state if possible, i.e. the first one is switched back to active and the remaining ones to standby, expressed by the maxsync operator. Since `l[1]` is able to switch to active and the repair unit is also able to follow via `R.done`, the `sync` operator can be used to ensure that the reaction is synchronous, i.e. all failed links are repaired at once.

```
R.rDone guards sync {
  R.⟨done⟩ , l[1].⟨swactive⟩,
  maxsync (i in{2 .. numLink}) { l[i].⟨swstandby⟩ }
}
```

The resulting network module definition is shown in the following listing. Herein the expand statement is used in order to abbreviate the instantiation of redundant links and the switching function which handles the link failures:

```
Module Network(numLink) : R←Repair {
  Behavior BLink {...}
  Module Link : BLink {...}

  Instance l[1] initially iActive of Link
  expand (i in{2 .. numLink}) {
    Instance l[i] initially iStandby of Link
    AND[j in{1 .. i−1}] l[j].cFailed & l[i].cStandby guards
      l[i].⟨swactive⟩
  }
  Condition nfailed = AND[i in{1 .. numLink}] l[i].cFailed

  l[1].cFailed guards R.⟨notify⟩
  R.rDone guards sync {
    R.⟨done⟩ , l[1].⟨swactive⟩,
    maxsync (i in{2 .. numLink}) { l[i].⟨swstandby⟩ }
  }
}
```

The system behaviour BSys consists of three states, i.e. sstandby, sfailed and shazard. Depending on the triggered guard labels <systemfail> and <recovered>, the behaviour can toggle between the states sstandby and sfailed. Furthermore, an (exponentially delayed) hazard transition is defined which can turn the system to the state hazard in case the safety function was temporarily unavailable. The listing of BSys is given as follows:

```
Behavior BSys {
  State sstandby, sfailed, shazard
  Transitions from sstandby
    if ⟨systemfail⟩ → sfailed
  Transitions from sfailed
    if ⟨recovered⟩ → sstandby
    if ⟨true⟩ → shazard, delay exponential 2.0
}
```

The root instance now has to be defined for the fault-tolerant-network. The pivotal element is the `System` keyword. As mentioned above, its meaning is twofold: Firstly, it is a module definition following the same syntactic rules as any other Module definition. Secondly, it defines the root instance, which is named `FTN` and inherits the behaviour `BSys`.

```
System FTN : BSys { ... }
```

The subsequent paragraphs explain the content of its body. Firstly, the processors' behaviour is specified followed by its Module definition. Next, the instantiation is provided including the interactions.

The processors' behaviour is very simple since it only comprises a Boolean behaviour, i.e. comprising two states, where, after an exponential delay, the processor fails from `pstandby` to `pfailed`.

```
Behavior BProcessor {
   State pstandby, pfailed
   Transitions from pstandby
     if ⟨true⟩ → pfailed, delay exponential 0.005
}
```

The Module definition `Processor` inherits the Behavior definition `BProcessor`. A state is set as the initial state by denoting an `Initial` statement and a Condition variable `failed` which is offered to its environment:

```
Module Processor : BProcessor {
  Condition failed = BProcessor.failed
  Initial good = BProcessor.good
}
```

Note that even such simple Behavior definitions have to be wrapped by a Module. Accordingly, direct Behavior instantiations (apart from their instantiation by inheritance) are currently disallowed by language definition.

It remains to define the subinstances of the system instance, i.e. two instances `p[1]` and `p[2]` of processors, wrapped by an expand statement iterating over a range `{1..2}`, and the network instance `n` parametrised by two links.

```
expand (i in {1 .. 2}) { Instance p[i] of Processor }
Instance n of Network(numLink=2)
```

An explicit initial state for the inherited behaviour of the system instance is denoted by

```
Initial standby = BSys.sstandby
```

Failures occurring inside subinstances have an impact on the system behaviour. The interaction dependencies can be modelled by Condition statements specifying the cause in terms of Boolean expressions on states, and guards-/forward statements specifying how the event influences the system in terms of a triggered behaviour.

A Condition representing the systems redundancy structure specifies the situation which will cause a system failure if one of the processors fails or the network fails:

```
Condition srs = OR[i in {1 .. 2}] p[i].pfailed | n.nfailed
```

The Condition is used inside a guards statement to describe the dependency between the system behaviour and the causal situation. In this case the Condition leads to a system failure:

```
srs guards BSys.⟨systemfail⟩
```

If the causal situation disappears (i.e. by a repair event) the system behaviour will recover:

```
!srs guards BSys.⟨recovered⟩
```

In the following the complete LARES FTN example model is provided. It abbreviates the fully exposed Module and Behavior definitions:

```
Behavior Repair  {...}
Module Network(numLink) : R←Repair {...}
Behavior BSys {...}
System FTN : BSys {
    Behavior BProcessor {...}
    Module Processor : BProcessor {... }

   expand (i in {1 .. 2}) { Instance p[i] of Processor }
   Instance n of Network(numLink=2)
   Initial standby = BSys.sstandby

   Condition srs = OR[i in {1 .. 2}] p[i].pfailed | n.nfailed

    srs guards BSys.⟨systemfail⟩
    !srs guards BSys.⟨recovered⟩
}
```

Finally, the questions of interest have to be specified for the model. The following probability measures are therefore stated inside the system definition of the FTN example in order to determine the probability of the system being in a specific state (i.e. `standby`, `failed` or `hazard`) or the probability of the network having failed:

```
Probability standby = Transient(BSys.sstandby, 50)
Probability failed = Transient(BSys.sfailed, 50)
Probability hazard = Transient(BSys.shazard, 50)
Probability nfailed = Transient(n.nfailed, 50)
```

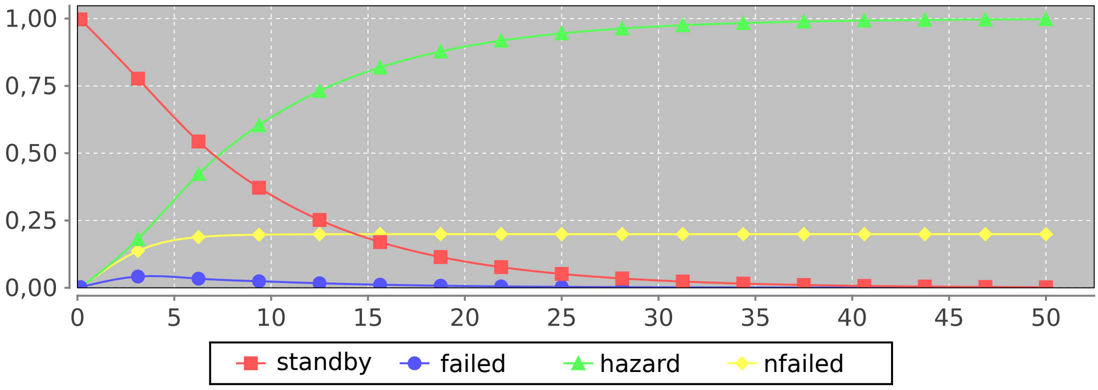


Figure 1.8: FTN: Analysis results of the basic model (hereby, the y-axis represents the fraction of the probability mass regarding a specified state and the x-axis represents the elapsed time)

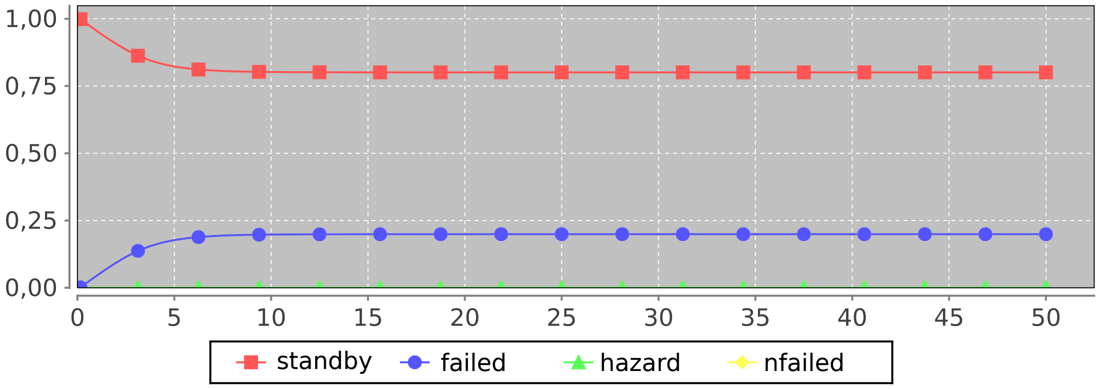


Figure 1.9: FTN: Hazards and processor failures disabled (hereby, the fraction of probability mass in the course of the elapsed time for the specified measure `failed` exactly overlays the values obtained for `nfailed`)

Based on the previously provided example model, the first analysis results are shown by Figure 1.8. It depicts that the probability of the network being unavailable stabilises due to the repair behaviour specified within the model. During the time in which the redundancy structure of the system is not fulfilled, hazards can occur. This leads to an aggregation of probability mass at the hazard state. The probability of the failed state is only considerably large before hazardous events deduct probability mass.

Other interesting analysis results can be obtained by varying some model parameter. Figure 1.9 represents the analysis results in case that the model is varied by two aspects: The first one being the hazard rate set to 0.0, i.e. no hazards can occur, and the second one being failure rate set to 0.0, i.e. no processor failures can occur.

```
Transitions from failed → hazardous, delay exponential 0.0
Transitions from good → failed, delay exponential 0.0
```

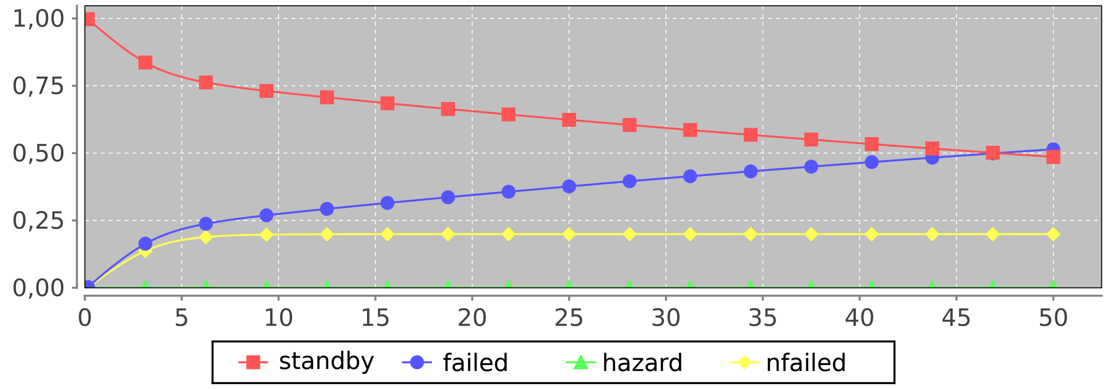


Figure 1.10: FTN: Hazards are disabled

One can see that due to the absence of a hazardous situation and the ability of the processors to fail while the network is repaired, the system availability, i.e. `FTN_standby`, converges. Note that the `FTN_failed` plot lies precisely on top of the `FTN_nfailed` probability. This means that for this model parametrisation, the probability of a system being in the `failed` state exactly corresponds to the probability of a network being failed.

Figure 1.10 represents the analysis results in case that the basic model is varied by setting the hazard rate to 0.0, thus meaning no hazards can occur. The figure illustrates that, due to the absence of a hazardous situation, the system availability, i.e. `FTN_standby`, decreases due to the processors' failure rates (at the same time at which the network is in a repaired state).

1.3 Component Monitoring System Example

In this section, a component monitoring system (MS) is considered which aims at assuring functional safety of the monitored systems, e.g. as defined by IEC 61508 [87], where monitoring for random errors during operation and safe incidence handling are claimed.

Let the monitoring system comprise two sensors for each monitored component (cf. Figure 1.11). Assume that a component consists of a light barrier and its power supply. Someone inside a safety cage cannot be detected and a hazardous situation might consequently not be prevented, whenever there is a power loss. Each component can be monitored by a sensor which can be substituted by another dedicated spare sensor in case of a failure. The redundancy introduced by the dedicated spare sensor for each component increases the monitoring reliability. In a situation where a redundancy substructure is not fulfilled (e.g. if both sensors of the parallel structure `S1` and `S2` as given by Figure 1.11 fail), at least one of the components is not monitored any more.

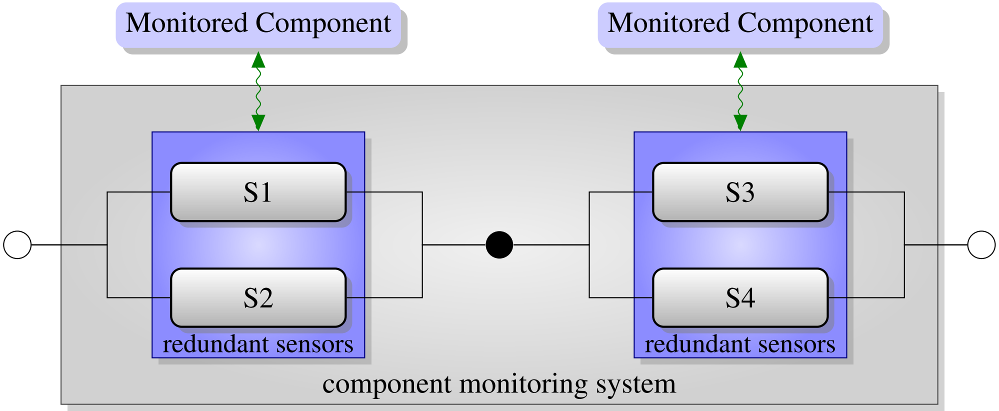


Figure 1.11: Redundant component monitoring system example

When a single sensor has failed, it may be safely substituted by the spare sensor for monitoring the other component. A reconfiguration can be performed which deploys the other spare sensor to monitor the currently unmonitored component. Apart from that, all sensors are switched to a mode `degr` with degraded performance in order to lower the failure rate and therefore extend a sensor's lifetime. The functioning of the system can thus be reestablished at the cost of some reconfiguration time. Meanwhile, when the reconfiguration takes place, the reconfigured sensor could fail as well: If that occurs, the system will definitely fail. Otherwise, the system can continue working in its second phase until one of the applied components fails. In Figure 1.12 the structure of the model is denoted as exemplified hereafter.

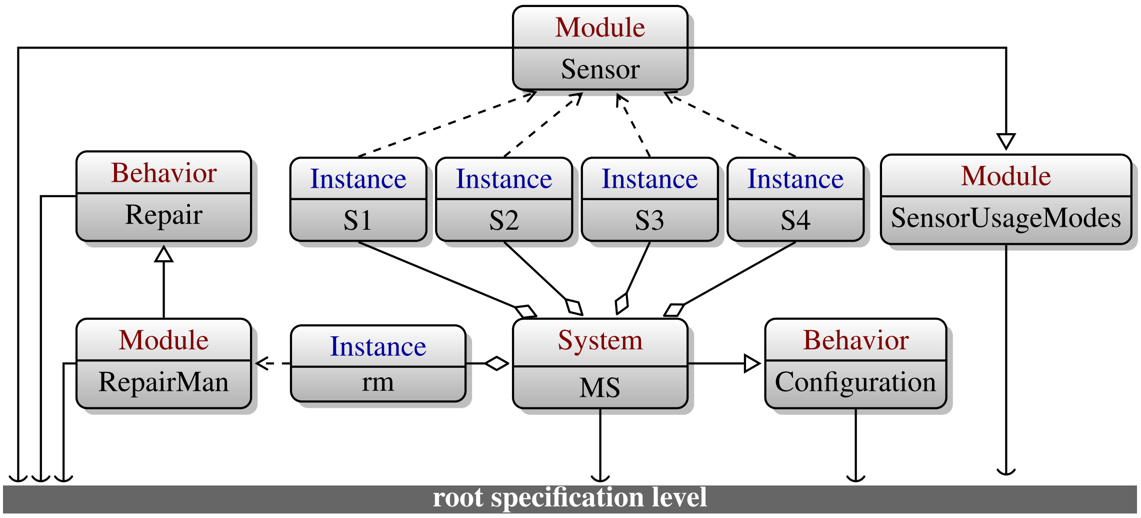


Figure 1.12: MS: The model structure

The Behavior definitions are not given in form of a textual LARES specification. Instead, they are depicted graphically by Figure 1.13. The main system behaviour is given in Figure 1.13(a). It comprises four states,

1. the initial phase (`RS1`), where the redundancy structure for the first phase holds,
2. the reconfiguration phase (`Reconf`), where the system tries to reconfigure if there is an unused working sensor
3. the second phase (`RS2`), where the reconfigured sensor is in use, and
4. the failure situation (denoted by `Fail`),

and a number of guarded and unguarded transitions representing

- the reaction that will be triggered by the guard label `<rs1f>` if the initial redundancy structure fails,
- or if the second redundancy structure is not fulfilled any more, the transition comprising the guard label `<rs2f>` will be triggered leading to `Fail`,
- the reconfiguration which takes a certain amount of time is determined by the rate of an exponential distribution ($\xrightarrow{1.0}$), or
- the repair or recovering transition `<rep>`.

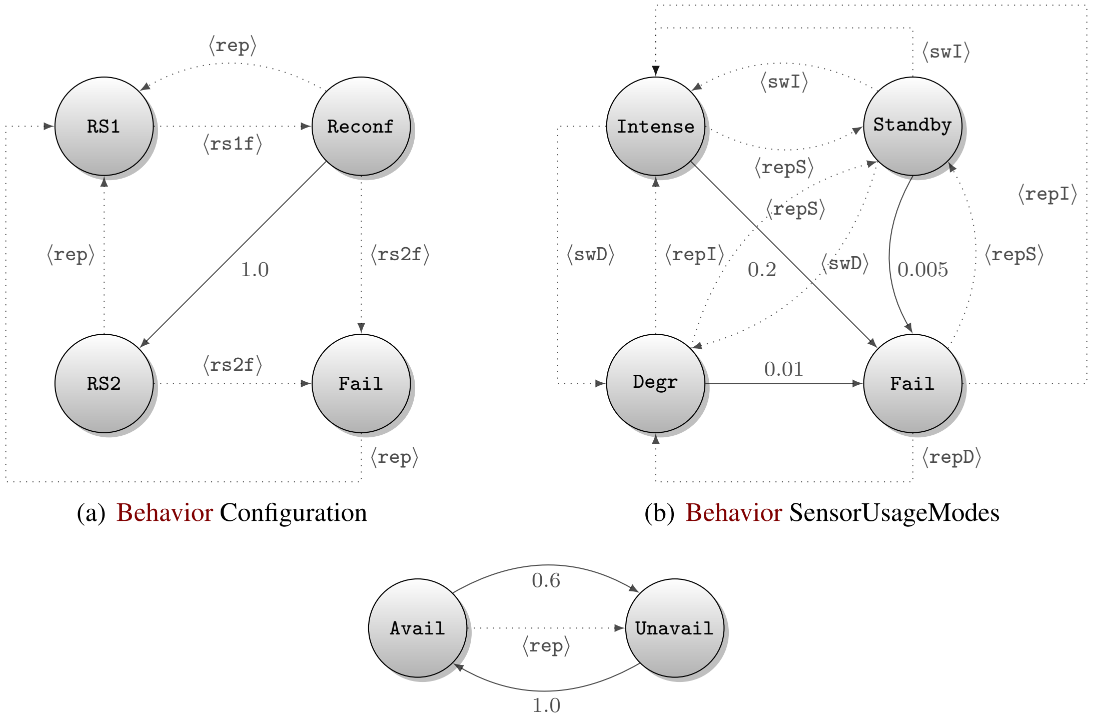
The sensor behaviour comprises the modes `Intense`, `Standby`, `Degr` and `Fail` (cf. Figure 1.13(b)). When a sensor is in its degraded mode `Degr` the failure rate of the sensor is less than in the `Intense` mode, e.g. due to reduced sensor frequency.

The inherent default failure behaviour is described by a number of exponential unguarded transitions: Failing from `Intense` by $\xrightarrow{0.2}$, from `Standby` by $\xrightarrow{0.005}$ and from `Degr` by $\xrightarrow{0.01}$. Furthermore, a number of switching and repair actions are defined using the guard labels `<swD>`, `<swI>`, `<repD>` and `<repI>`.

Finally, a repair behaviour is defined (cf. Figure 1.13(c)) in which a repair can toggle between available (`Avail`) or unavailable (`Unavail`) over the events defined by $\xrightarrow{0.6}$ and $\xrightarrow{0.5}$. The guard label `<rep>` would be triggered if the current state was `Avail`.

The sensor module inherits from `SensorUsageModes` by `SUM <- SensorUsageModes` defining the abbreviated name `SUM` which is subsequently used to address the inherited behaviour:

```
Module Sensor : SUM ← SensorUsageModes { ... }
```



(a) Behavior Configuration

(b) Behavior SensorUsageModes

(c) Behavior Repair

Figure 1.13: MS: The Behavior definitions

In the body of this Module definition two initial configurations `I` and `S` are offered to the environment. The instantiated behaviour `SUM` is initially set to the `Standby` mode in case that `S` is addressed otherwise `SUM` starts in the `Intense` mode:

```
Initial I = SUM.Intense
Initial S = SUM.Standby
```

A number of forward statements make the guard labels of the inherited Behavior accessible to be triggered by external events:

```
forward ⟨swI⟩ to SUM.⟨swI⟩
forward ⟨swD⟩ to SUM.⟨swD⟩
forward ⟨repI⟩ to SUM.⟨repI⟩
forward ⟨repS⟩ to SUM.⟨repS⟩
```

To provide the opportunity for an environment to reason about the local state, a `Condition` serves as an interface aggregating state information, which means for this case, that good abstracts the set of states `Intense`, `Standby` and `Degr`:

```
Condition fail = SUM.Fail
Condition good = not SUM.Fail
```

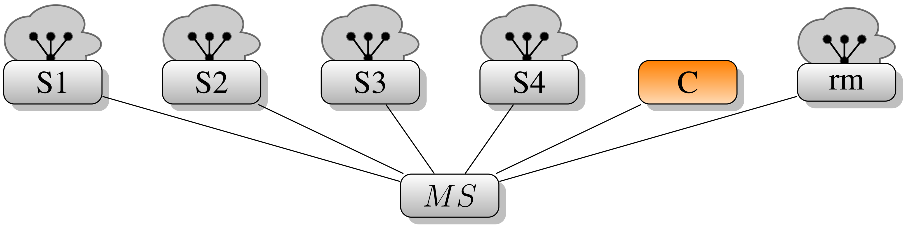


Figure 1.14: MS: The instance tree of the model

A repair unit inheriting the repair behaviour is also assigned to the system. It only provides a single Initial configuration, i.e. being unavailable. Whenever the internal behaviour is able to perform `<rep>`, the environment is able trigger `<rep>` introduced by the contained forward statement:

```
Module RepairMan : Repair {
  Initial U = Repair.Unavail
  forward ⟨rep⟩ to Repair.⟨rep⟩
}
```

The final definition to be specified is the root instance `MS` of the system. It inherits the `Configuration` behaviour abbreviated by `C`:

```
System MS : C←Configuration { ... }
```

Each subinstantiation inside the body of `MS` specifies a name, an initial configuration and an addressed type of Module:

```
Instance S1 initially I of Sensor
Instance S2 initially S of Sensor
Instance S3 initially I of Sensor
Instance S4 initially S of Sensor
Instance rm initially U of RepairMan
```

The resulting instance tree of Module instantiation including the inherited system behaviour instance `C` is shown in Figure 1.14. In this example, the indicated subtree on top of the Module instances only comprises a single Behavior instance.

Next, the interaction among the instances is defined by guards statements. If sensor `S1` fails, the standby mode of `S2` will then be activated, or, similarly, if sensor `S3` fails, its standby mode of `S4` will then be activated:

```
C.RS1 & not S1.good guards S2.⟨swI⟩
C.RS1 & not S3.good guards S4.⟨swI⟩
```

If the initial safety structure is no longer fulfilled while being in state $RS1$, such that not at least one out of two sensors are working for each of component (hence, the operator `oo` is defined), the system will try to reconfigure itself, in such form that the system behaviour

C synchronously performs the behaviour triggered by <rs1f> together with the maximal synchronised behaviour of all the sensors. Those that can switch to its degraded mode by allowing triggering <swD> will synchronously perform their switching behaviour:

```
C.RS1 & not (
  1 oo {S1.good, S2.good} & 1 oo {S3.good, S4.good}
) guards sync{
  C.⟨rs1f⟩, maxsync{S1.⟨swD⟩, S2.⟨swD⟩, S3.⟨swD⟩, S4.⟨swD⟩}
}
```

If the initial redundancy structure has already failed, i.e. not C.RS1 and the reconfigurable redundancy structure is not satisfied any more by the remaining sensors, where at least two out of all sensors have to be good, the system behaviour will be triggered by <rs2f>.

```
not C.RS1 & not (2 oo {S1.good, S2.good, S3.good, S4.good}) guards
  C.⟨rs2f⟩
```

Finally, the assigned repair strategy is to take action merely in case that the system has already reached its failed state. This will be achieved by trying to synchronously transfer all subinstances of the system back to their initial state, except those that still are in their initial state (expressed by the maxsync operator of the sensors).

```
C.Fail guards sync {
  C.⟨rep⟩, rm.⟨rep⟩,
  maxsync{S1.⟨repI⟩, S2.⟨repS⟩, S3.⟨repI⟩, S4.⟨repS⟩}
}
```

In order to be able to perform an analysis, a number of measures are introduced in the system definition body. The states of interest are those defined by the system behaviour.

```
Probability fail = Transient(C.Fail, 50)
Probability reconf = Transient(C.Reconf, 50)
Probability RS1 = Transient(C.RS1, 50)
Probability RS2 = Transient(C.RS2, 50)
```

Corresponding to the given measures, transient analysis is performed up to 50 time units including 15 intermediate calculation steps as shown in Figure 1.15.

Also, complex states defined by a Condition statement can be used within a Probability statement:

```
Condition a = C.RS1 | C.RS2
Probability working = Transient(a, 50)
```

The analysis of the measures is given in Figure 1.15. It shows that the probability of RS1 decreases very quickly because of the sensors being used in the intense mode at the beginning. Thereby increasing the probability of being in Reconf. Of course, in the

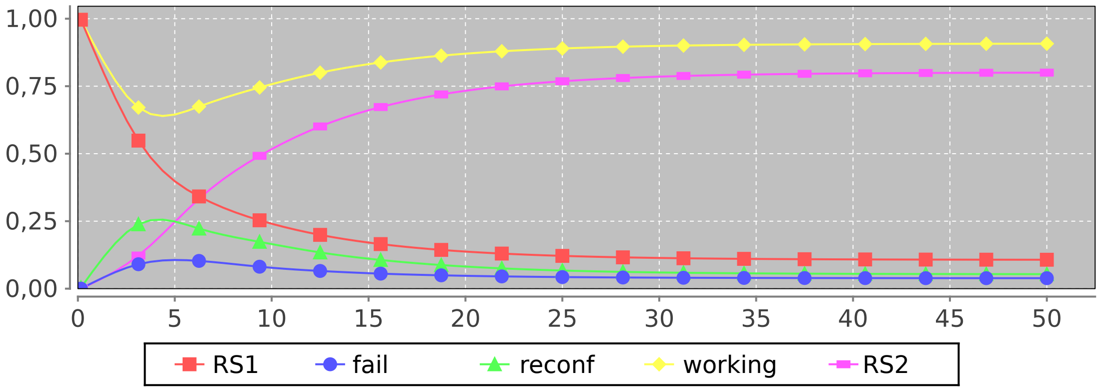


Figure 1.15: MS: Basic model

way the probability of RS1 decreases, the probability of RS2 increases. The remaining probability mass congregates in the Fail state. The yellow curve represents the working probability of being in RS1 and RS2.

In the following section the contribution to the domain of dependability modelling and analysis will be discussed.

1.4 Contribution

The LARES language definition aims at encouraging users to construct reusable components (thereby defining their own dependability concepts in terms of generic LARES constructs rather than being limited to specialised features that other languages offer). As a consequence thereof, a modeller is commonly not required to extend the language semantics himself. In case that requirements for the analysis of a model change (in such way that also the semantics of LARES have to be extended or that only a restricted subset of LARES has to be addressed), the language and the whole infrastructure (which supports the language, the transformation and the analysis) are designed in the perspective of extensibility to ease future adaptations and extensions.

Many approaches do not disclose language formalisation, in particular the corresponding semantics. This may be due to some (business) culture or because there is no formalisation beyond code. The contribution of this work is to create confidence by exhaustively unveiling the formal language definition in terms of syntactical rules and its abstract representation. The complete formalisation of the underlying semantics is given in terms of stochastic process algebra and by means of labelled transition systems. To emphasize the uniqueness of the language semantics, a formal proof shows a correspondence between the transition-system-construction rules given by the LTS semantics and the rules that build an SPA specification which itself is based on semantic rules which map to a transition system.

Even a complete formalisation does not prevent developers from making programming errors, it merely helps to reduce these errors. Hence, diversification by implementing several transformations based on different rules to produce the same output help identify these errors. The contribution of the LARES framework is that two transformations which either apply rules to construct an LTS directly or construct a corresponding SPA specification, are implemented. An equivalence check of the final transition system reveals almost all occurring semantic issues and programming errors. This approach helps to further increase confidence in the LARES approach in general.

1.5 Organisation of this Thesis

So far, the LARES language has been motivated, informally introduced and exemplarily illustrated. Furthermore, its contribution to the domain of dependability modelling and analysis has been discussed. The reader has been given a compendium of the important language features and the way these are used. It is required to convey a formal definition of the language syntax and its semantics to describe how the defined concepts and transformations are captured by an implementation in order to feature a larger case-study and to discuss the relation of LARES and its concomitant implementation to other approaches. Thus, the topics that will be addressed in this thesis are featured below:

In Chapter 2, a formal definition of the syntax is granted by denoting an abstract representation of the language elements and the concrete syntax. Chapter 3 refers to the abstract formal representation of the language. It is structured by foremost starting off with a transformation into a subset of LARES, and afterwards subsequent transformations will be defined into some targeted formal languages. Those languages are supported by tools which enable users to analyse the models and to determine the measures of interest. Two transformations are provided in this chapter: One addresses SPA and the other transforms into a labelled transition system (LTS). Since the semantics of the SPA itself is given by means of LTS, both transformations are defined such that, when applied to a model, their final result is behaviourally equivalent and thus define equal semantics for LARES. The chapter is concluded by providing the theoretical basis to validate those transformations against each other and to check for inconsistencies in the implementation or finding other flaws. Furthermore, a formal proof is sketched, showing the equivalence between two transformations into different target formalisms. In connection with the theoretical foundations of the language, the transformation semantics and its validation, Chapter 4 considers the technical aspects of the implementation. Chapter 5 focuses on case-studies. Especially one larger case-study will emphasize the applicability of LARES by showing that it scales well and the learning curve remains moderate. Finally, Chapter 6 compares related languages and contrasts LARES, while Chapter 7 concludes the work.

Chapter 2

LARES - Formal Language Definition

In the introductory chapter an informal view on LARES was given and illustrated with the help of two example systems. This chapter provides an exhaustive formalisation of the language in terms of both an abstract mathematical representation and a concrete syntax conforming to EBNF (Extended Backus-Naur Form). The chapter begins with a section introducing the subsequently used notation. Afterwards, each language element is detailed by formally denoting its set theoretic representation and the associated EBNF rules representing the concrete syntax. Readers who would preferably like to make use of the exact language definition or semantics (as given in Chapter 3) are strongly encouraged to read through this chapter to pick up the subsequently applied definitions and notations. Frequently used tuple notations are provided for future reference in Appendix A.

2.1 Notation

Standard set theoretic notation [51] is predominantly used in this work, such as union $\cup$, intersection $\cap$, set difference $\setminus$ and Cartesian product $\times$. The notation $M = \{x \mid \varphi(x)\}$ is a set builder which denotes a set M containing all those elements x of some specified domain where a certain statement $\varphi(x)$ is satisfied. In addition to these standard notations, less common notations are revisited or introduced in this section.

Multisets

Multiple occurrences of elements within a group of objects might be addressed by tuples. Due to the fixed order, indices have to be used to access a specific object. Dealing with lots of indices is difficult and many cases are conceivable where no order is required. That leads to the definition of multisets as introduced in [2, 127, 128]. A multiset behaves in a way similar to a set and can be defined as a tuple consisting of a basic set of elements and a cardinality function providing the multiplicity of each element. The notation $[a, a, b, c]$ is equivalent to $[a^2, b^1, c^1]$, where the multiplicity is given by the superscript. Let $M \in mset(X)$ denote a multiset of objects of the set X, $m \in^k M$ then states that M contains exactly k instances of elements m. A builder notation as it is defined for sets can also be used to construct a multiset by elements x with cardinality k: $M = [x^k \,|\, \varphi(x, k)]$. Operators on multisets are as straightforward as they are for sets, e.g. $M \cup N$ is defined by the union of the basis sets of M and N and the sum of their cardinalities.

The Sequence, Power Set, Option and Either Types

A tuple $x \in X^n$ is always of fixed length n. A sequence is introduced as a tuple of arbitrary (finite) length with elements from M. The set of all finite sequences over M is defined as

$$Seq(M) = \{(a_1, ..., a_m) \mid m \in \mathbb{N}, a_1, ..., a_m \in M\} \cup \{()\}$$

It holds that $Seq(M) = \bigcup_{m \in \mathbb{N}} M^m$ and $M^0 = \{()\}$. In this work, the $\circ$ operator is commonly used as a composition operator. The composition of two sequences is defined by their concatenation. Thus, the concatenation function

$$\circ\colon Seq(M) \times Seq(M) \to Seq(M)$$

is defined as $(a_1, ..., a_k) \circ (b_1, ..., b_l) = (a_1, ..., a_k, b_0, ..., b_l)$. Obviously $M^k \circ M^l = M^{k+l}$ holds for $k, l \in \mathbb{N}$. The power set is defined in the standard way, as the set of all subsets:

$$\mathcal{P}(X) = \{M \mid M \subseteq X\}$$

An *option* contains an element (i.e. $m = 1$) of a set M or is empty (i.e. $m = 0$)

$$Opt(M) = \bigcup_{m \in \{0,1\}} M^m$$

Naming Convention and Projection of Attributes

To facilitate the access to the attributes of a tuple, an abbreviating notation is introduced. Let $\mathcal{X} = \prod_{i \in L} A_i$, where $\prod$ denotes the crossproduct of the sets A_i and L is a set of indices. Additionally, let a tuple $x \in \mathcal{X}$ be represented by $(a_1, \ldots, a_{|L|})$. Each projection $\pi_i : \mathcal{X} \to A_i$ () maps to the i-th element of x such that $\pi_i(x) = a_i$. As an example, let $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{D}$ be some arbitrary, but not explicitly defined sets which are now used to define the universal set $\mathcal{X} = \mathcal{A} \times \mathcal{P}(\mathcal{A}) \times \mathcal{B} \times Seq(\mathcal{D})$. An element $x \in \mathcal{X}$ is denoted as a tuple of attributes (a, S_A, b, L_D). Each attribute is represented by an associated label where a denotes an element of $\mathcal{A}$, S_A denotes a subset of $\mathcal{A}$, b denotes an element of $\mathcal{B}$ and L_D denotes a list/tuple of elements of $\mathcal{D}$. To access these attributes, their names are used to denote their projection functions, e.g. for the attributed tuple $x = (a, S_A, b, L_D)$, the projection

- $x.a = \pi_1(x)$ maps to the 1st element of x,
- $x.S_A = \pi_2(x)$ maps to the 2nd element of x,
- $x.b = \pi_3(x)$ maps to the 3rd element of x and
- $x.L_D = \pi_4(x)$ maps to the 4th element of x.

The notation as it is used here is related to the tuple calculus as described in [48].

Capturing Recursive Tree Structures

As shown in Section 1.1, LARES is a language that is able to describe hierarchy. Thus, abstract language elements have to be defined which allow constructing finite recursive structures. Let $\mathfrak{E}$ denote a recursive element. The following statement has to hold:

$$x \in \mathfrak{E} \Leftrightarrow \exists n \in \mathbb{N} : x \in \mathfrak{E}_n$$

It means that x can only be a valid model corresponding to $\mathfrak{E}$ once it is contained by a universal set of structures of depth n (i.e. $x \in \mathfrak{E}_n$). The set of hierarchies follows the abstracted representation:

$$\mathfrak{E}_n = \mathcal{P}(\bigcup_{i=0}^{n-1} \mathfrak{E}_i) \quad \text{for } n \geq 1 \text{, where} \quad \mathfrak{E}_0 = \{\emptyset\}$$

$\mathcal{P}$ may also be substituted by Seq if the substructure requires an order or other mixed forms. To give an example, an arbitrary tree structure which corresponds to the above hierarchy is depicted. In contrast to Knuth's trees [93] which grow downwards, this tree grows upwards as shown in Figure 2.1. All leaf nodes are valid recursive elements since they correspond to $\mathfrak{E}_0$. Each substructure node is a valid element since it is contained by the universal set of structures of depth $n \in \mathbb{N}$.

Exceeding the pure structure, additional elements $x \in X$ can be assigned to each node inside the hierarchy:

$$\mathfrak{E}_n = \mathcal{P}(\bigcup_{i=0}^{n-1} \mathfrak{E}_i) \times X \quad \text{and} \quad \mathfrak{E}_0 = \{\emptyset\} \times X$$

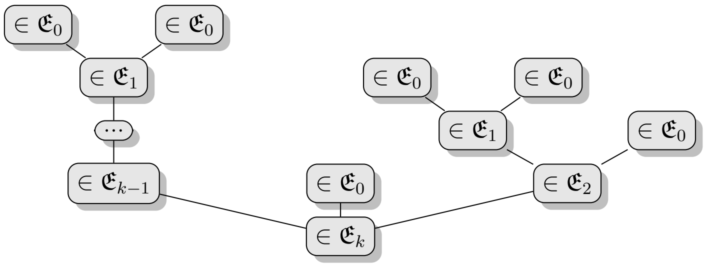


Figure 2.1: A LARES tree structure

From now on, recursive definitions will always underlie this scheme to ensure a sound set theoretic formalisation. For the reader's convenience, the indices are neglected. For this purpose, let $:=$ be used to denote recursive definitions (as they appear due to the recursive nature of Module definitions, Instance statements, expand statements and several common sublanguage definitions) in order to implicitly define hierarchic structures. To give an example, let the universal set of Module definitions be denoted by $\mathcal{M}$. The following definition may be given to specify the recursive nature of Module definitions. Hereby,

$$\mathcal{M} := \mathcal{P}(\mathcal{M}) \times X \qquad \textbf{is synonymous to}$$

$$\mathcal{M} = \bigcup_{i=0}^{\infty} \mathcal{M}_i, \text{ where } \mathcal{M}_n = \mathcal{P}(\bigcup_{i=0}^{n-1} \mathcal{M}_i) \times X \text{ and } \mathcal{M}_0 = \{\emptyset\} \times X$$

meaning that, in this case, the set $\mathcal{M}$ is defined as the union of all inductively defined tree structures of arbitrary depth $\bigcup_{i=0}^{\infty} \mathcal{M}_i$ with an assigned element $x \in X$ at each node.

2.2 Concrete and Abstract Syntax of LARES

As it was announced in Chapter 1, an exhaustive formalisation of the LARES language will now be provided with the help of the notations introduced in the preceding section. The formal definitions will be twofold. On the one hand, the abstract syntax is specified using set-theoretic and first-order predicate calculus notation. And on the other hand, the concrete syntax is given by a sort of EBNF grammar based on the Xtext language [58] as used for the LARES language implementation of the editor. It is an LL(k) grammar [92] used by an LL parser in order to perform the recursive descent constructing the leftmost derivations. To prevent infinite descent, left-recursive rules are disallowed.

2.2.1 The LARES Root Element

A LARES model consists of a System definition together with an arbitrary number of Module and Behavior definitions. The universal set of LARES models is hence given by

$$\mathcal{LARES} := \mathcal{P}(\mathcal{B}) \times \mathcal{P}(\mathcal{M}) \times \mathcal{I},$$

where $\mathcal{B}$ denotes the universal set of Behavior definitions (cf. Section 2.2.2), $\mathcal{M}$ denotes the universal set of Module definitions and $\mathcal{I}$ denotes the universal set of instances (cf. Section 2.2.3). Accordingly, a tuple $(B, M, i_1) \in \mathcal{LARES}$ denotes a LARES specification. Hereby, B is the set of Behavior definitions, M is the set of Module definitions and $i_1 \in \mathcal{I}$ is the system instance which the parser *resolved* in such a way that a Module definition $m_s \in M$ is already associated.

The concrete syntax will now be specified. The internal naming of the elements of the abstract syntax tree (AST), which result from the application of the grammar rules, usually corresponds to the notation of the abstract representation. For example, let `B` denote all Behavior definitions parsed by stating `B+=BehaviorDefinition`, the corresponding notation of the abstract representation is hence given by B. Hereby, the operator `+=` adds each abstract representation obtained by an application of the rule `BehaviorDefinition` to `B`. The multiplicity of a rule's application is specified by the operators `+`, `*` or `?`, as known from regular expressions.

The first rule which parses a LARES specification is deduced from the following, future demands on LARES. Based on the future requirement to import other LARES specification files that come without a specific System definition (i.e. serving like an imported model library), the AST is constructed by using the given rule without distinguishing the System definition from the other plain Module definitions:

```
LaresSpec : (B+=BehaviorDefinition | M+=ModuleDefinition)*;
```

The single System instance as denoted by the abstract formal representation therefore has to be extracted while parsing and is hence not explicitly captured by the concrete syntax.

Identifier

Each abstract definition can be parametrised: The parameters which are statically defined in the context of an instantiated definition, can be used (apart from serving as rates or weights) inside expand statements or other expandable expressions (i.e. all expressions which allow iterator expressions to be used). A number of resembling statements or expressions (inside the expandable expressions) can be abbreviated by using index variables. This of course requires a subsequent evaluation using the dependent parameters in order to obtain the expanded expressions. For this purpose, indexed identifiers are introduced. They can be used for *named statement*s (e.g. either by serving as a name for a *named statement* or by referring to a *named statement*). The universal set of all indexed identifiers is defined as

$$\mathfrak{ID} := \Sigma^{+} \times \mathit{Seq}(\mathfrak{AE}),$$

where Σ^{+} represents the set of valid labels and $\mathfrak{AE}$ denotes the set of integer-valued arithmetic expressions (defined in Section 2.2.4). An identifier can thus be denoted as a tuple $(l, I) \in \mathfrak{ID}$, where l is a label and $i \in I$ is an index which appears in form of an arithmetic expression. The concrete grammar rule for specifying identifiers which may include index variables is

```
ID : l = ident (’[’ I+=AE (’,’ I+=AE)* ’]’)?;
```

The concrete grammar keyword `ident` does hereby represent the parsing rule for all standard identifiers, whereas Σ^{+} denotes the associated abstract representation.

Reference

Let the set of namespaces be given by $\mathcal{NS} := \mathit{Seq}(\mathfrak{ID})$. The universal set of references to a named statement is then defined by

$$\mathfrak{Ref} := \mathcal{NS} \times \mathfrak{ID},$$

where the tuple $(i, l) \in \mathfrak{Ref}$ denotes the reference i (addressing an instance via its namespace) and l (addressing a named statement). The concrete grammar is given as follows:

```
Ref : : i+=ident ’.’ l=ident
```

Parameter Expression

The Module and the Behavior definition can be parametrised. The universal set of parameters are defined by

$$\mathfrak{PE} := \mathcal{P}(\mathfrak{P}),$$

where $\mathfrak{PE}$ denotes all sets of parameters $\mathfrak{P} := \Sigma^{+} \times \mathit{Opt}(\mathfrak{AE})$ consisting of a label l and an element e whose default value may optionally be specified in terms of an arithmetic expression. The concrete grammar is determined by

```
PE : P (’,’ P)*;
P : l=ident (e=AE)?;
```

Definition Reference

The tuple $(r, p) \in \mathfrak{Ref}_{\mathfrak{D}} := \Sigma^+ \times \mathfrak{PE}$ represents a reference to a definition. Note that for resolving a reference, the referred definition has to be *in scope*. Its concrete grammar is given by

```
RefD : r=ident (’(’ p=ParamExpr ’)’)?;
```

2.2.2 Behavior Definition

This section will formalise Behavior definitions, whereas the subsequent section is going to detail Module definitions. Since both Behavior and Module definitions employ expand statements, two types of expand statements have to be defined.

Each of them may only contain statements which coincide with those allowed by the context. For this reason, the formal definition of a Behavior or a Module definition is split up, separating the body of statements in order to prevent redundancy in the language definition.

Due to its simplicity, the Behavior definition is denoted at first. A Behavior definition is an automaton comprising states and transitions. It is obvious that each definition requires a name. The preceding chapter anticipated that each definition can be parametrised and the body has to be separated to avoid definition redundancy when it comes to defining expand statements. As a consequence, the universal set of Behavior definitions $\mathcal{B}$ is defined as

$$\mathcal{B} := \Sigma^+ \times \mathfrak{PE} \times \mathfrak{B}_{body} \times \mathfrak{Ref}.$$

A Behavior is a tuple $(l, p, b, ic) \in \mathcal{B}$, where l denotes an unindexed label, p denotes the parameter expression, b denotes the body of the Behavior definition and ic is the initial configuration, which refers by default to the first occurring state of the body. The concrete syntax does not explicitly require denoting the initial configuration, as the default configuration is either implicitly defined by the first occurring state or it is derived during the model instantiation which will be explained later. The corresponding syntax rule is thus given by

```
BehaviorDefinition: l=ident ("(" p=ParamExpr ")")? b=BehaviorBody ;
```

As a next step, the Behavior body is defined. It serves as a container for statements. These statements define the desired sequential behaviour terms of an automaton (consisting of states and transitions). A body allows including an arbitrary number of expand statements, so that each one recursively contains a further Behavior body. The universal set of Behavior bodies is thus defined as

$$\mathfrak{B}_{body} := \mathcal{P}(\mathcal{S}) \times mset(\mathcal{T}) \times \mathcal{P}(\mathfrak{B}_{expand})$$

A behaviour body $b \in \mathfrak{B}_{body}$ is denoted as a tuple (S, T, E), where S is the set of states, T is the multiset of transitions and E is the set of expand statements for a Behavior body. The sets of elements for $\mathcal{S}$, $\mathcal{T}$ and $\mathfrak{B}_{expand}$ will be defined in the subsequent sections. The corresponding concrete grammar rule applies further rules. These allow a modeller to specify a number of states within a single expression or to specify several transitions starting from a single source state at once. For this reason, the sS attribute of the following rule denotes a set of State statements and msT denotes the multiset of Transitions statements which in turn is a multiset of outgoing transitions.

```
BehaviorBody: "{"
  (sS+=StateStmt | msT+=TransitionsStmt | E+=ExpandBehaviorStmt)*
"}";
```

In the following sections all statements allowed inside a Behavior body are formally introduced. Note that the syntax rules StateStmt and TransitionsStmt construct sequences of states and transitions respectively. The union of all contained elements contained by sS or msT yields a datatype which corresponds to S or T respectively.

The State Statement

In order to start with the simplest element of the abstract representation, the named statement State is introduced which is represented by a single identifier. The universal set of State statements is thus defined as:

$$\mathcal{S} := \mathfrak{I}\mathfrak{D}$$

In contrast, the concrete grammar allows modellers to specify a number of states within a single State statement by denoting a comma-separated list of names:

```
StateStmt: "State" S+=State ("," S+=State)*;
State: name = ID;
```

The `Transitions` Statement

Following Chapter 1, several kinds of transitions can be defined within a Behavior definition, i.e. *unguarded transitions* which were not assigned any guard label and *guarded transitions* for whom an explicit guard label was defined. Each transition has a source and a target statement. A transition statement may also contain information about the timing distribution of that transition. The set of transitions is given by

$$\mathcal{T} := \mathcal{S} \times Opt(\mathfrak{ID}) \times Opt(\mathcal{D}) \times \mathcal{S}.$$

A transition $t \in \mathcal{T}$ is denoted by the tuple (s, g, d, t), where s refers to a source state, g may contain a guard label, $d \in Opt(\mathcal{D})$ may contain a distribution and t refers to the target state. Note that $\mathcal{D} := \mathcal{D}_{delayed} \cup \mathcal{D}_{imm}$. Thus, a distribution $(type, value) \in \mathcal{D}$ is either delayed by following a delay time distribution contained by $\mathcal{D}_{delayed}$, or immediate, by abiding to a discrete distribution with an assigned weight as an element of $\mathcal{D}_{imm}$. Both $\mathcal{D}_{delayed}$ and $\mathcal{D}_{imm}$ will be introduced subsequently. The concrete grammar is split into two separate rules:

```
TransitionsStmt: "Transitions" "from" s = ID (transitions += Transition)+;
Transition: ("if" (’<’ g=ID ’>’)? "->" t=ID ("," d=Distribution )?;
```

Hereby, the first rule parses a source state of a number of transitions arising from the iterative application of the second rule. The second rule constructs transitions which consist of the parsed source state, a target state, the assigned distribution and an optionally defined guard label. An `if <true>` is equivalent to the case in which no guard label `g` is defined. If no distribution is explicitly specified, it will depend on the context of the specification which form the interpretation will ultimately take: The distribution type predetermined by its interaction partner will then be implicitly chosen. If there is no partner, the default distribution type will be immediate with weight 1. A specified distribution is either exponential (assigned with a rate) or else immediate (assigned with a weight). The syntactical rule is defined by

```
Distribution: ("delay" Exponential) | Immediate;
```

If the `delay` keyword is present, the `Exponential` rule will be applied. Otherwise, the `Immediate` rule will take action. The distribution types are captured by the set $D_{types} = \{\texttt{immediate}, \texttt{exponential}\}$. The set of immediate distributions are defined as $\mathcal{D}_{imm} := \{\texttt{immediate}\} \times \mathfrak{AE}$, where each element contains a weight attribute. The corresponding concrete syntactical rule is

```
Immediate: "weight" w=AE;
```

Currently, the set of delaying distributions $\mathcal{D}_{delayed}$ solely encompasses exponential distributions. It is defined as $\mathcal{D}_{delayed} := \mathcal{D}_{exp}$, where $\mathcal{D}_{exp} := \{\texttt{exponential}\} \times \mathfrak{AE}$. The concrete syntax for a delay expression and the exponential distribution is

```
Exponential: "exponential" r=AE;
```

The contained arithmetic expressions for the rate or the weight will later be evaluated to a value of the domain of positive floating point numbers (denoted by $[\![\mathfrak{AE} : \mathbb{R}^{\geq 0}]\!]$).

Remark: In [138] other non-exponential probability distributions (e.g. deterministic or Weibull distributed firing times), timer variables and rules to modify them were proposed. The probability and the duration of a *timed transition* (i.e. a transition that is attributed with a timer variable) is determined by the current value of a timer. However, these aspects are currently not part of LARES.

A set of transitions $T \subseteq \mathcal{T}$ can be partitioned depending on the existence of a defined guard label into the set of guarded transitions T_g and the set of unguarded transitions T_u:

$$T_g := \{t \in T \mid t.g \neq \{()\}\} \text{ and } T_u := T \setminus T_g$$

A guarded transition $(s, (g), d, t) \in T_g$ or an unguarded transition $(s, (), d, t) \in T_g$ can be further distinguished by the currently available distributions d. Different notations are provided for these cases:

	$d = (\texttt{exponential}, \lambda)$	$d = (\texttt{immediate}, w)$
$(s, (g), d, t)$	$s \xrightarrow{\langle g \rangle, \lambda} t$	$s \overset{\langle g \rangle, w}{\dashrightarrow} t$
$(s, (), d, t)$	$s \xrightarrow{\lambda} t$	$s \overset{w}{\dashrightarrow} t$

The definition of guarded transitions underlies a restriction: All guarded transitions within a Behavior definition that provide the same guard label must have an equal distribution type. Otherwise, a Behavior definition is invalid. This restriction was introduced in order not to complicate the SPA transformation, as a significantly larger amount of pieces of local information would be required.

The `expand` statement inside a `Behavior` definition

An expand statement enables a modeller to easily replicate statements within a body. Instead of writing a number of similar or repetitive statements, an abbreviated notation can be made by applying the expand statement. Figure 2.2 shows an example in which, instead of explicitly defining three states inside a Behavior's body, the expand statement is used to replicate its inner statement by a variable `i` that iterates over a given *set expression*

```
State  s[1]                expand (i in {1..3}) {
State  s[2]       ⟹           State  s[i]
State  s[3]                }
```

Figure 2.2: Use of an expand statement

$\{1..3\}$ called *iterator expression*. The set of expand statement for a Behavior definition is defined as

$$\mathfrak{B}_{expand} := \mathfrak{IE} \times \mathfrak{B}_{body},$$

where $(ie, b) \in \mathfrak{B}_{expand}$ denotes an expand element tuple containing an iterator expression ie, which is detailed in Section 2.2.4, and a Behavior body b. The corresponding grammar rule is given by

```
ExpandBehavior: "expand" ie=IE b=BehaviorBody;
```

2.2.3 Module Definition

As it was in Chapter 1, a Module definition may inherit from visible Behavior or Module definitions by the principle of *Delegation*. For each delegate an alternative name can be specified. A Module definition resembles a Behavior definition by the fact that it can be parametrised and allows expand statements to be employed inside its body. Apart from inheritance, a Module definition differs from the Behavior definition by providing means for the instantiation of subcomponents and the definition of interactions among the contained instances.

To illustrate a Module definition, a snippet from the FTN example introduced in Section 1.2 is used. There, a Module `Network` is defined by a single undefined parameter `numLink` and an inherited Behavior definition `Repair` comprising the alternative name `R`. The Module body is not detailed for reasons of brevity:

```
Module Network ( numLink ) : R ← Repair {...}
```

In accordance with the preceding example, the formal representation of the set of Module definitions is

$$\mathcal{M} := \Sigma^{+} \times \mathfrak{PE} \times \mathfrak{DE} \times \mathfrak{M}_{body},$$

where a tuple $(l, p, d, b) \in \mathcal{M}$ denotes the name l of the module, the parameter expression p, the delegate expression d and the Module's body b. The concrete grammar is given by

```
ModuleDefinition: l=ident ("(" p = DefinitionParameters ")")?
        (":" d = Delegates)? b=ModuleBody;
```

Following the approach to implement inheritance by the principle of *Delegation*, the set of *delegate expression*s is captured by

$$\mathfrak{DE} := \mathcal{P}(\mathcal{I})$$

Hereby, each delegate expression may capture a number of references. Each reference to an abstract definition will later instantiate a *Delegate*. For a sound specification of multiple delegates having identical definition types, the ability to set a unique name is required.

In order to encompass all instantiations of Module and Behavior definitions ($\mathcal{I} := \mathcal{I}_{\mathcal{B}} \cup \mathcal{I}_{\mathcal{M}}$), the set of Behavior instantiations $\mathcal{I}_{\mathcal{B}}$ and the set of Module instantiations $\mathcal{I}_{\mathcal{M}}$ are defined:

$$\mathcal{I}_{\mathcal{B}} = \mathfrak{ID} \times (\mathfrak{Ref}_{\mathcal{B}} \cup \mathcal{B}) \times Opt(\mathfrak{Ref}) \quad \text{and} \quad \mathcal{I}_{\mathcal{M}} = \mathfrak{ID} \times (\mathfrak{Ref}_{\mathcal{M}} \cup \mathcal{M}) \times Opt(\mathfrak{Ref}).$$

A single instance is denoted by a tuple $(l, t, ic) \in \mathcal{I}$, where l is the unique identifier of the instance to be constructed from a definition reference t, where $t.r$ is the label of the definition that is parametrised by $t.p$, and ic may contain an initial configuration. After resolving the reference type (cf. Section 3.3.1), the reference is substituted by the parametrised definition such that either $t \in \mathcal{B}$ or $t \in \mathcal{M}$ holds. The following two rules define the concrete grammar of delegate expressions:

```
Delegates: D += Delegate ("," D += Delegate)*;
Delegate: ( u=ID "<-")? RefD ("initially" ic=Ref)?;
```

The name of the referenced definition will implicitly be the unique identifier u if it is not explicitly differently stated by the model specification. Moreover, a reference to an initial configuration (e.g. referring to a state or an Initial statement) can be declared optionally.

Remark: It might be a future desire to abbreviate the model description such that all interface statements (i.e. the *named statements*, such as State, Condition and forward statements or defined *guard labels*) of each delegate instance can be directly addressed. For this purpose, implicit referable statements which yield a one-to-one mapping to these interface statements over the current Module instance have to be introduced. If there is a name clash by inheriting from two definitions having equally named statements, a mapping is required to uniquely address the interface statements of the delegates in order to settle this conflict. This feature is currently not implemented in LARES because it comes with concerns over the modularity of a model (an interface would depend on the inner structures and thus the substitution of inner elements might change the interface).

Finally, the Module body $b \in \mathfrak{M}_{body}$ is formally defined. It may contain Behavior or Module definitions, statements for instantiations, for interactions among the instances and the initial configuration. The universal set of Module bodies is thus given by

$$
\begin{aligned}
\mathfrak{M}_{body} :=\ & \mathcal{P}(\mathcal{B}) \times \mathcal{P}(\mathcal{M}) \times \\
& \mathcal{P}(\mathcal{I}_{\mathcal{M}}) \times \mathcal{P}(\mathfrak{C}) \times mset(\mathcal{G}) \times \mathcal{P}(\mathcal{F}) \times Seq(\mathfrak{IC}) \times \mathcal{P}(\mathfrak{M}_{expand}) \times \mathcal{P}(\mathfrak{Prob}).
\end{aligned}
$$

A body is denoted as $(B, M, I, C, G, F, IC, E, P) \in \mathcal{M}_{body}$, where B is the set of Behavior and M the set of Module definitions. Both of them will be visible within the subtree defined by the body, unless they are overwritten by other definitions with equal names. Moreover, I is the set of Instance statements which determine the Module substructure exceeding the one constructed by the delegates. C contains the set of Condition statements, the multiset G contains the guards statements and F represents the set of forward statements by which the interaction among the instances and delegates can be described. Finally, IC is a list (allowing the prioritisation) of Initial configurations, E is the set of expand statements defined at the body layer of the module, and P is the set of `Probability` statements. The concrete grammar rule for the module body is provided by:

```
ModuleBody:
  body?="{" (
    B+=BehaviorDefinition | M+=ModuleDefinition | I+=InstanceStmt |
    C+=ConditionStmt | G+=GuardStmt | F+=ForwardStmt |
    IC+=InitialStmt | E+=ExpandModuleStmt | P+=ProbabilityStmt
  )* "}";
```

In the following, the types of statements contained by Module bodies will be formalised.

The Instance Statement

Each module may define a number of subinstantiations, which ultimately implies an instance tree. In the beginning of Section 2.2.3, the abstract representation of Module instantiations $\mathcal{I}_{\mathcal{M}}$ is defined, which includes the instantiation of delegates. The first definition is now straightforwardly used to represent Module instantiations by Instance statements. In contrast to an instantiation of a delegate, an explicit definition of an instance name is mandatory. Equally, the parametrisation will have to be specified if that is required by the Module definition. Furthermore, an initial configuration may explicitly be defined which directly refers to a State of a Behavior instance or to a specific Initial statement offered by the Module to be instantiated. Else, in the case of absence of an explicit addressing of an initial configuration, the implicit default Initial, i.e. the first occurring one, will be taken. The associated concrete EBNF rule is given by

```
InstanceStmt:
  "Instance" l=ID "of" t=ModuleReference ("initially" ic=Ref)?;
```

A *named statement* (such as an Initial, Condition and forward statement) inside the assigned module definitions of a subinstances can be addressed by providing the instance name and the name of the statement. The initial configuration specified by the `initially` keyword refers to an Initial statement. As the Instance statement already defines the name of the instance, the reference to the Initial statement is therefore not allowed to bear the name of the instance. The reference does not explicitly denote the name of the instance, as this information is implicitly given by the Instance statement.

The Condition Statement

A Condition statement assigns a name to a *condition expression* which may refer to other Condition statements (that are either locally defined or by the associated module definition of a child instance) or to states (which are provided via instantiated delegates). Condition expressions can be translated such that they only address state variables provided by the instantiated behaviours at the leaves of the instance tree. They will do so by resolving the indirections via references to Condition statements as illustrated by Figure 2.3.

As it was previously mentioned, the referred conditions can either be in the local scope or referenced indirectly over an instance. A state variable can be referred via an instantiated behaviour. Referring to arbitrary levels inside a subtree is not allowed. Except from referring to local conditions (belonging to the local layer), it is solely possible to refer to

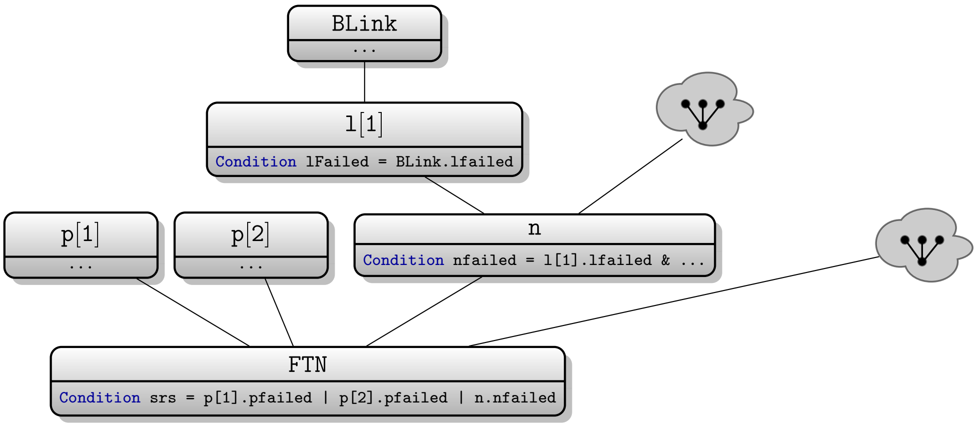

BLink
...
l[1]
Condition lFailed = BLink.lfailed
p[1]
...
p[2]
...
n
Condition nfailed = l[1].lfailed & ...
FTN
Condition srs = p[1].pfailed | p[2].pfailed | n.nfailed

Figure 2.3: FTN: Indirections to states of Behavior instances via references to Condition statements (depicted along the instances `FTN`, `n`, `l[1]` and `BLink`)

states of the Behavior instances belonging to the local delegates or to Condition statements provided by local subinstances (belonging to the adjacent layer of child instances).

In Section 2.2.4, the universal set of condition expressions $\mathfrak{CE}$ will be defined. A condition expression $\mathfrak{CE}$ is a Boolean expression, extended by specific atomic elements referring to states or to Condition statements. The universal set of Condition statements can be captured by

$$\mathfrak{C} := \mathfrak{ID} \times \mathfrak{CE}$$

A single statement is represented by the tuple $(l, c) \in \mathfrak{C}$, where l denotes the name of the condition and c its assigned condition expression. The concrete grammar is given by:

```
ConditionStmt: "Condition" l=ID "=" c=ConditionExpression;
```

The guards Statement

Apart from the informal description given in Section 1.1, a guards statement may not only be composed of a single *generative condition* expression (enabling the guards statement to trigger a reaction) and a *reactive* expression (specifying the reactive behaviour to be triggered). Instead of a single reactive expression, it may also comprise a number of so called *conditional reactives*. A *conditional reactive* is a reactive expression that will only take action if a certain *restrictive condition* is satisfied.

An example guards statement comprising a number of *conditional reactives* is given to illustrate this concept:

```
A.a guards {
    sync{A.⟨b⟩, B.⟨b⟩} if !A.x & B.x
    B.⟨b⟩ if !A.y
    A.⟨b⟩
}
```

The statement has a *generative condition* A.a and three *conditional reactives*. This means that if instance A satisfies some Boolean variable referenced by a, a reaction will be evoked that was predetermined by the three *conditional reactives*. These *conditional reactives* may constitute a choice between the possible reactions described by each *conditional reactive*.

		!A.x&B.x	
		true	*false*
!A.y	*true*	sync{A.⟨b⟩, B.⟨b⟩} + B.⟨b⟩ + A.⟨b⟩	B.⟨b⟩ + A.⟨b⟩
	false	sync{A.⟨b⟩, B.⟨b⟩} + A.⟨b⟩	A.⟨b⟩

Table 2.1: Arising choices between several reactions depending on the given restrictive conditions

A restrictive condition is specified inside the first two *conditional reactive*, whereas the third one is always $true$. These restrictive conditions hereby induce 2^2 cases shown in Table 2.1. They arise due to the combinations of $true/false$ evaluations of the restrictive conditions. As the third *conditional reactive* is always satisfied, it does not contribute to the number of combinations. As a consequence, the reactions induced by the reactive expression `A.`$\langle$`b`$\rangle$ are always eligible, whereas it depends on the evaluation of the conditions, whether the reactions induced by `sync`$\{$`A.`$\langle$`b`$\rangle$`,B.`$\langle$`b`$\rangle\}$ or `B.`$\langle$`b`$\rangle$ are competing with `A.`$\langle$`b`$\rangle$. If both restricting conditions are fulfilled, each of their assigned reactive expressions will imply some ways to behave which are in choice to each other and to `A.`$\langle$`b`$\rangle$. A choice between a number of reactive expressions is subsequently denoted by the operator $+$.

The reactive expression may apply different kinds of operators such as maxsync, sync or choose, where each imposes a special (informally described) semantics:

- maxsync: *„The transitions of those guard label variables that can be satisfied will react synchronously.“*
- sync: *„All addressed guard label variables have to be satisfied such that the associated transitions react synchronously.“*
- choose: *„Exactly one addressed guard label variable is satisfied such that an associated transition reacts.“*

Whenever the actual reactive behaviour differs from the evaluation of the required behaviour specified by the reactive expression, no reaction will arise.

Note that a reactive expression will be translated to a Boolean expression later on. Hence, some properties are exemplified below

- As LARES is a descriptive language, all operators are commutative, e.g. sync(a,b,c) is equivalent to sync(b,a,c).
- Due to the Boolean logic interpretation of sync and maxsync by $\wedge$ and $\vee$ respectively, both operators are associative.
- Furthermore, sync and maxsync are distributive over a choice. For illustratory purposes, let $+$ denote a choice between a and b. The term sync(a+b,c) is then equivalent to sync(a,c) + sync(b,c).
- The n-ary choose operator cannot be expressed in terms of Boolean logic by a composition of binary xor operators (as xor is not associative). To overcome this obstacle, an explicit translation semantics will be defined in an upcoming section of this work.

Furthermore, other interesting properties (e.g. that choose is not distributive over maxsync) are not further detailed here. Nevertheless, the choose operator has to be used with care due to its non-associativity property.

Let $\mathfrak{RE}$ denote the set of reactive expressions (as defined afterwards in Section 2.2.4). A guards statement can be formally denoted as a tuple (g, CR, ns), where $g \in \mathfrak{CE}$ denotes the *generative condition* and $CR \in \mathfrak{CR}$ is a multiset of *conditional reactive*s, whose universal set is given by $\mathfrak{CR} := multiset(\mathfrak{CE} \times \mathfrak{RE})$. Using a multiset instead of a usual set is important since multiple identical conditional reactive may be specified. Hereby, the namespace attribute ns is only required to later identify the origin of a guards statement. The universal set of guards statements is thus given by

$$\mathcal{G} := \mathfrak{CE} \times \mathfrak{CR} \times \mathcal{NS}$$

Each *conditional reactive* $cr \in CR$ can be represented by the tuple (c, r), where c denotes the condition and r denotes the *reactive part*. The corresponding grammar rule is

```
GuardStmt: g=CE "guards" ( r=RE_atom | ("{" (CR+=CndReact)+ "}"));
```

Note that this rule allows specifying a single reactive expression or a list of *conditional reactive*s. A *conditional reactive* is thus implicitly constructed in which the *restrictive condition* c is `true` and r is set by the specified reactive expression to match the abstract syntax. The grammar rule for a *conditional reactive* is given by

```
CndReact: r=RE_atom ("if" c=CE)?;
```

The forward Statement

Since the visibility restriction only allows users to refer to the adjacent levels, forward statements are introduced as a mechanism to forward the reactive triggering events to the instantiated behaviours. A forward statement defines a *forward label* which can be used in the same way as a *guard label* of a transition and can hence be triggered by a reference within a reactive expression. A triggering event will be considered if the auxiliary global condition of the forward statement and at least one *restrictive condition* of its *conditional reactive*s are satisfied. Depending on these conditions, different reaction arise. A forward is similar to a guards statement and therefore the universal set of forward statements is defined as

$$\mathcal{F} := \mathfrak{CE} \times \mathfrak{ID} \times \mathfrak{CR}.$$

A single forward is represented by a tuple $(c, l, CR) \in \mathcal{F}$, where c denotes the global auxiliary *generative condition* and CR is the set of *conditional reactive*s (see the definitions

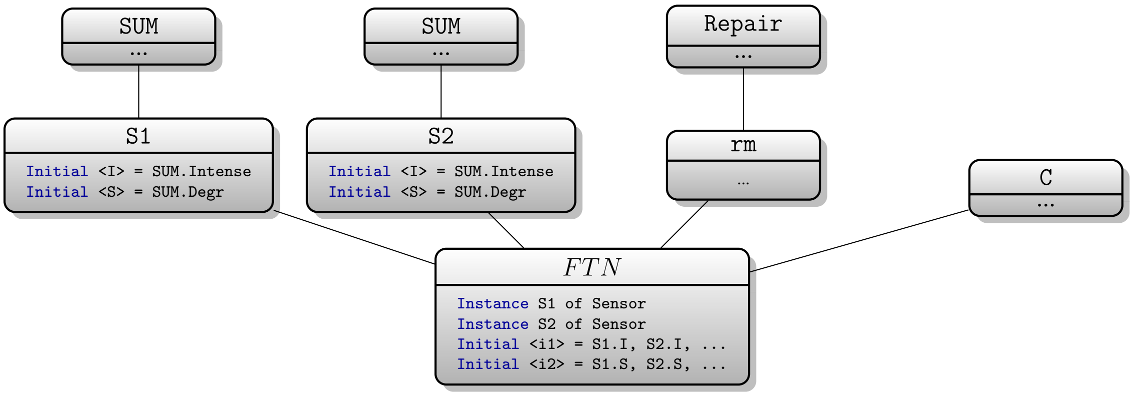


Figure 2.4: Modified MS example: Providing initial configurations

as given for the guards statement). It differs from a guards statement by comprising a forward label l. The corresponding concrete grammar is defined by the following rule:

```
ForwardStmt:
  ('if' c=CE)? 'forward' l=ID ( r=RE | ("{" (CR+=CndReact)+ "}"));
);
```

This rule constructs the abstract syntax tree by implicitly assuming that a *conditional reactive* has a $true$ condition in case that only a single reactive expression instead of an explicit list of *conditional reactive*s is provided.

The Initial Statement

Each Behavior instance needs to be provided with an initial configuration. A Behavior's initial configuration may, apart from the locally defined initial state, be configured at arbitrary level within the model hierarchy. Therefore the Initial statement is introduced to the language.

This statement gives a modeller the opportunity to specify several Initial statements inside each Module definition in order to provide a choice between different initial configurations. An initial configuration either refers to states of Behavior instances or to Initial statements provided by the child instances. In case of addressing a specific Initial statement of a Module instantiation, its configuration is recursively propagated into the instance tree. The overall goal is to ultimately provide an initial state for each instantiated behaviour at the leaves of the instance tree of the model.

The example illustrated in Figure 2.4 shows the FTN definition which provides two Initial statements. If i1 is chosen for the instantiation of FTN, the subinstances S1 and S2 will be configured by the Initial statement I. By that, Intense is set as the initial state for the associated behaviour instance SUM for both instances. Otherwise, if i2 had been addressed, the associated behaviours would ultimately have been instantiated with Degr as their initial state.

The universal set of Initial statements is given by

$$\mathfrak{IC} := \mathfrak{ID} \times \mathcal{P}(\mathfrak{Ref}).$$

An Initial statement can then be represented by the tuple $(l, IC) \in \mathfrak{IC}$ comprising a label l and a set of references IC referring to either another Initial statement within a module instance or a State statement within an instantiated behaviour. The concrete grammar is denoted as follows:

```
InitialStmt: ("Initial" l = ID)? ("=") IC += Ref ("," IC += Ref)* ;
```

Note that if no explicit reference to an Initial statement for a Module instance is defined, the first one specified inside the Module will be default. If no Initial statements are defined, it will lie in the responsibility of a Module's subinstances to specify their (default) initial configuration. Since a Behavior instance always defines a default initial state which may be overridden by an external configuration, a complete initial configuration of a system is guaranteed.

The Probability Statements

In essence, a LARES model is specified in order to determine quantitative measures of the modelled system. As LARES deals with stochasticity, the language provides mechanisms to denote desired probability measures. Both steady state and transient state probability measures can be defined:

$$\mathfrak{Prob} := \mathfrak{Prob}_S \cup \mathfrak{Prob}_T, \text{ where } \mathfrak{Prob}_S := \mathfrak{ID} \times \mathfrak{CE} \text{ and } \mathfrak{Prob}_T := \mathfrak{ID} \times \mathfrak{CE} \times \mathfrak{AE}$$

A steady state probability measure can hence be represented by a tuple $(l, c) \in \mathfrak{Prob}_S$ and a transient probability measure can be represented by a tuple $(l, c, t) \in \mathfrak{Prob}_T$. Hereby, a measure denotes a name l for the measure and specifies a set of states by a *conditional reactive* c. The transient analysis method will be used if a time interval t is given, otherwise the steady state analysis will be performed. The concrete grammar rule is given by

```
ProbabilityStmt:
  "Probability" name=ID "="
  (type="Transient" "(" c=CE "," t=AE ")") |
  (type="Steadystate" "(" c=CE ")");
```

The expand Statement

An expand statement enables modellers to easily generate repetitive statements within a body. Similar to what has been shown for the body of a Behavior definition, a Module body also provides an expand statement as a means to avoid writing a number of similar repetitive statements. The universal set of expand statements inside module bodies is given by

$$\mathfrak{M}_{expand} = \mathcal{P}(\mathfrak{IE}) \times \mathfrak{M}_{body},$$

where $(ie, b) \in \mathfrak{M}_{expand}$ represents a single expand statement.

The abstract representation of the sets of iterator expressions $\mathfrak{IE}$ is formally defined in Section 2.2.4. Each iterator evaluation defines a parameter that is used to generate the repetitive statements comprising parametrisable identifiers specified in the body of the Module. The associated grammar rule is given by

```
ExpandModuleStmt: "expand" iters=IE b=ModuleBodyRestr;
```

For modularity reasons and due to the fact that Module and Behavior definitions are already parametrisable, the environmentally defined variables are not required to have an impact on the definition. For this reason, the concrete grammar rule explicitly allows the body of an expand statement to contain statements (i.e. Condition, forward, guards, Instance, Initial, Probability and expand), whereas it disallows the specification of further Module and Behavior definitions:

```
ModuleBodyRestr:
  body?="{" (
    I+=InstanceStmt |
    C+=ConditionStmt | G+=GuardStmt | F+=ForwardStmt |
    IC+=InitialStmt | E+=ExpandModuleStmt | P+=ProbabilityStmt
  )* "}";
```

2.2.4 Common Sublanguages and their Adaption to LARES

Within the preceding sections, several expressions were used that have not been formalised yet. All of these can be seen as common sublanguages embedded within LARES. They can be separated into different groups and will be detailed in the subsequent sections:

- Arithmetic expression: Used in parameter expressions or within set expressions
- Set expression: Used within expand statements or some atomic elements of Boolean expressions or reactive expressions

- Iterator expression: Used to construct parametrised statements of evaluations to avoid writing similar lines of code; used by expand statements and by operators within condition and reactive expressions
- Boolean expression: An abstract Boolean expression from which specialisations can be derived, e.g. condition and reactive expressions
- Condition expression: A Boolean expression which is specialised by atomic elements referring to state variables, as used in Condition statements, *conditional reactive*s or as *generative condition* within guards statements
- Reactive expression: A Boolean expression which is specialised by atomic elements to specify the propagation along the paths through the instance tree, thereby addressing some required behaviour to be executed simultaneously

Arithmetic Expression

Arithmetic expressions are extensively used within LARES. A leaf element might either refer to a label of a parameter defined inside a definition or be an explicit value within the domain of integer or floating point numbers. The distinction between integer-typed and floating-point numbers is omitted for reasons of convenience and subsequently subsumed by real numbers $\mathbb{R}$. Furthermore, the standard operators are used to perform calculations. Within this section, the formal definition of the abstract and concrete syntax of arithmetic expressions as used within LARES is specified.

For this purpose, a number of syntactic categories are defined, i.e. $a \in \mathfrak{AE}$, $n \in \mathbb{R}$ and $x \in \mathfrak{ID}$. The arithmetic expression language as used in LARES applies a number of binary operator symbols $\mathcal{OP} = \{+, -, *, /, \%\}$: $+$ for addition, $-$ for subtraction, $*$ for multiplication, $/$ for division and $\%$ for the modulo operation.

An arithmetic expression $a \in \mathfrak{AE}$ is defined as follows:

$$a := n \,|\, x \,|\, (a_1 + a_2) \,|\, (a_1 - a_2) \,|\, (a_1 * a_2) \,|\, (a_1/a_2) \,|\, (a_1\%a_2) | (a) \text{ , where } a_1, a_2 \in \mathfrak{AE}$$

Each expression $a \in \mathfrak{AE}$ may be composed of two arithmetic expressions using one of the above operators or else some atomic element, i.e. a number $n \in \mathbb{R}$ or some variable $x \in \mathfrak{ID}$ which will be replaced by a certain value or arithmetic expression later-on.

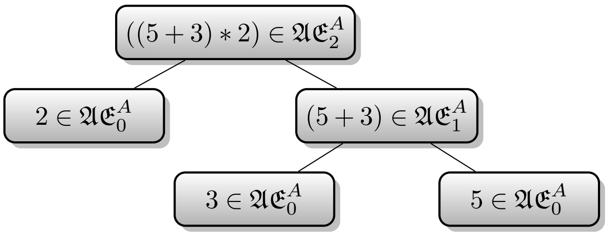
When considering the atomic elements $A = \mathbb{R} \cup \mathfrak{ID}$, the universal set of arithmetic expressions $\mathfrak{AE}$ (parametrised by A) unites all recursive structures defined by

$$\mathfrak{AE}_n^A = \bigcup_{i=0}^{n-1} \mathfrak{AE}_i^A \times \mathcal{OP} \times \bigcup_{i=0}^{n-1} \mathfrak{AE}_i^A \text{ for } n \geq 1, \text{ where } \mathfrak{AE}_0^A = A$$

Following the notation defined in Section 2.1, the preceding definitions are used to denote arithmetic expressions in LARES:

$$\mathfrak{AE} := (\mathfrak{AE} \times \mathcal{OP} \times \mathfrak{AE}) \cup A$$

For example, a valid abstract representation of an arithmetic expression $((5+3)*2)$ is represented by

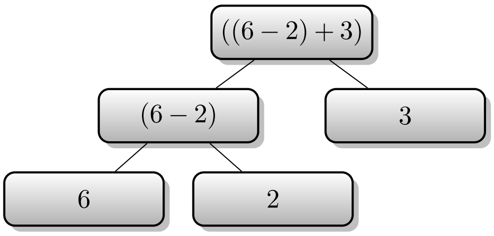

$((5+3)*2) \in \mathfrak{AE}_2^A$
$2 \in \mathfrak{AE}_0^A$
$(5+3) \in \mathfrak{AE}_1^A$
$3 \in \mathfrak{AE}_0^A$
$5 \in \mathfrak{AE}_0^A$

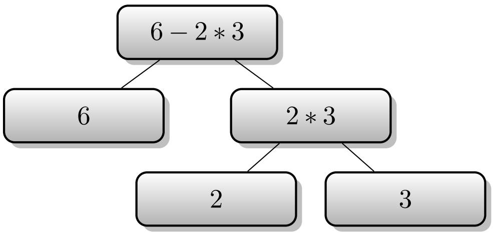
An arithmetic expression has to be decomposed in order to be evaluated. The attributed tuple representation for an element $a \in \mathfrak{AE}_n^A$, where $n > 0$, is given by (l, op, r) which is composed of a left-hand-side operand l, a right-hand-side operand r and an arithmetic operator op. If $a \in \mathfrak{AE}_0^A$, the element a will be either a number or an identifier.

The concrete syntax of an arithmetic expression is defined as follows. The grammar is split up into several rules in order to prevent left-recursion:

```
AE: l=AE2 (ops+=(’+’|’-’) R+=AE2)*;
AE2: l=AE3 (ops+=(’*’|’/’|’%’) R+=AE3)*;
AE3: ’(’ AE ’)’ | AEAtom
AEAtom: ID | Numeral
```

The order of the above rules defines the precedence. Hereby, parentheses have priority over e.g. `*`, and `*` has priority over `+`. The EBNF metasyntax allows stating repetitions within one line by the `*` operator. Thus, all operator/operand definitions with the same precedence defined at the same hierarchy level of the arithmetic expression are captured within `ops` and `R`. The corresponding binary tree structure mapping, e.g. for the arithmetic expression $6-2+3$, goes from left to right in contrast to $6-2*3$:





Set Expression

Set expressions are used in the iterator expressions of LARES to specify operations on sets of elements. A set expression $s \in \mathfrak{SE}$ is defined by

$$s = \mathfrak{SE}_{atom} \,|\, (s_1 + s_2) \,|\, (s_1 \setminus s_2) \,|\, (s_1 * s_2) \text{ , where } s_1, s_2 \in \mathfrak{SE}.$$

Set expressions are defined by a set of atomic elements and inductively by expressions composed of a binary operator and two sub-expressions. Hereby $+$ denotes the union, $*$ represents the intersection of two sets and $\setminus$ is the set difference.

Let the universal set of Boolean (infix) operators for set expressions be $\mathcal{OP} = \{+, *, \setminus\}$. The universal set of set expressions is defined by applying the notation provided in Section 2.1 in order to capture the inherent recursive nature of the expression:

$$\mathfrak{SE} := \mathfrak{SE}_{atom} \cup (\mathfrak{SE} \times \mathcal{OP} \times \mathfrak{SE})$$

Hereby, $\mathfrak{SE}_{atom}$ represents the atomic elements of a set expression. In LARES, specific specialisations of the atomic elements have been defined: A set expression comprises atomic leaf elements such as arithmetic expressions as well as elements specifying a certain range.

$$\mathfrak{SE}_{atom} := \mathfrak{SE}_{range} \cup \mathfrak{SE}_{set}, \text{ where } \mathfrak{SE}_{range} = \mathfrak{AE} \times \mathfrak{AE} \text{ and } \mathfrak{SE}_{set} = \mathcal{P}(\mathfrak{AE})$$

The concrete syntax of a set expression is as follows. In order to prevent left-recursion, the grammar is split up into several rules:

```
SE: l=SE2 (op=("+","*","\" R+=SE2)*;
SE2: '(' SE ')' | SEAtom
SEAtom: SESet | SERange
SESet: '{' elem+=AE (',' elem+=AE)* '}'
SERange: '{' start=AE ".." end=AE '}'
```

Hereby, the order of rules define the precedence (which allows users to omit explicit parenthesis, so that they can introduce an order). Union, intersection and set difference operators of set expressions are of equal order in terms of precedence. Thus, the use of parentheses is the only safe way to denote a specific order regarding the operations. An expression such as given by $\{1..3\} * \{2..5\} \setminus \{2\}$ is evaluated from left to right as shown in Figure 2.5.

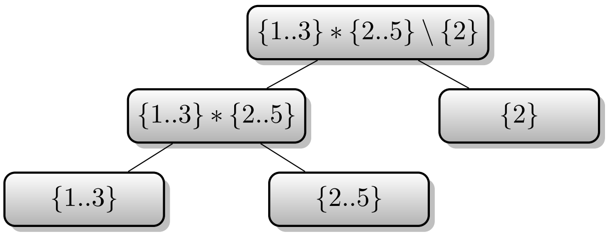


Figure 2.5: Set Expression Evaluation

Common Iterator Expression

In LARES, iterator expressions are used in expressions that expand their body which is composed of other expressions. Hereby, an *iterator* iteratively defines a parameter which is used for each evaluation of the associated body.

To illustrate the use of iterator expressions, two examples are extracted from the models presented in Chapter 1:

```
AND (i in {1 .. numLink}) l[i].cFailed
expand ( i in {1 .. 2 } ) { Instance p[i] of Processor }
```

A more sophisticated expression comprising several iterators within one expression is denoted by

```
AND (i in {1 .. 3}, j in {i ... 3}) l[i,j].cFailed
```

A set of tuples arises by the cross-product of the given set expressions by evaluating the above statements, so that each tuple (i, j) represents a combination of parameter definitions which are iteratively used to evaluate the body. Regarding the given example, the set of tuples is given by

$$(i, j) \in \{(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)\}.$$

A single iterator expression consists of a set of iterators. To give a formal notation, the universal set of iterator expressions can be represented by the set $\mathfrak{IE} := \mathcal{P}(\mathfrak{I})$. The universal set of iterators is given by $\mathfrak{I} := \{(l, s) \in \mathfrak{ID} \times \mathfrak{SE}\}$, where each iterator is composed of a variable identifier l and a set expression s. The concrete syntax is given by

```
IE:   '(' I+=Iterator (',' I+=Iterator)* ')';
Iterator:   l=ID "in" s=SE;
```

Common Abstract Boolean Expression

A Boolean expression is a type of propositional calculus in which atomic elements represent Boolean variables and a set of logical operators (e.g. `&`, `|`, etc) can be used for binary composition of arising subexpressions. The evaluation of such an expression maps to a Boolean value, as expected by its name.

In LARES, Boolean expressions are used to make assertions on the state of a (sub)system. Each of them may serve as a criterion for a subsequent reaction and is thus called *generative condition* expression. Depending on the context of their usage, an atomic element either corresponds to a specific state of an instantiated behaviour or to another proposition variable defined by another Condition. If a certain state specified by a *generative condition* expression is reached, a number of addressed reactions will be simultaneously triggered. These reactions may also be restricted.

It is another sort of application when a *reactive* expression for forward or guards statements is specified: For this purpose, a Boolean expression is specialised by specific atomic expressions as leaf nodes. These atomic expressions are of recursive nature and provide specific operators. The abstract representations of forward and guards statements are defined in order to capture these reactive (Boolean) expressions. The LARES grammar, by contrast, only allows defining a single atomic reactive expression in the beginning, for reasons which will be given detailed information on later in this thesis.

A Boolean expression $b \in \mathfrak{BE}^A$ (parametrised by some atom-type A) is defined as follows:

$$b := a \,|\, (b_1|b_2) \,|\, (b_1\&b_2) \,|\, (\neg b_3)\text{ , where } b_1, b_2, b_3 \in \mathfrak{BE}^A$$

The given abstract syntax denotes atomic elements $a \in A$, binary infix operators ($\&$ and $|$) and a unary negation operator ($\neg$).

The universal set of Boolean expressions $\mathfrak{BE}^A$ represents the union of all recursive structures defined by

$$\mathfrak{BE}^A_n = \left(\bigcup_{i=0}^{n-1} \mathfrak{BE}^A_i\right) \times \{|,\&\} \times \left(\bigcup_{i=0}^{n-1} \mathfrak{BE}^A_i\right) \;\cup\; \{\neg\} \times \bigcup_{i=0}^{n-1} \mathfrak{BE}^A_i\text{, where } \mathfrak{BE}^A_0 = A$$

Following the notation defined in Section 2.1, this definitions can equivalently be stated as

$$\mathfrak{BE}^A := \mathfrak{BE}^A \times \{|,\&\} \times \mathfrak{BE}^A \quad \cup \quad \{\neg\} \times \mathfrak{BE}^A \quad \cup \quad A$$

A left recursive definition is again not allowed in order to specify the concrete syntax definition. It is therefore split up in a number of rules that also define the precedence of its application:

```
BE: l=BE2 (ors+=BE_or)*;
BE_or: ('|' | 'OR' | 'or') r=BE2;

BE2: l=BE_atom (ands+=BE_and)*;
BE_and: ('&' | 'AND' | 'and') r=BE3;

BE3: (neg?=('!' | 'NOT' | "not"))? (e=BE_atom | "(" e=BE ")")
```

The hereby defined precedence priority for each operator is given by () > ! > & > |. Note that BE_atom still remains to be defined to the specific purpose depending on the context.

Remark: To distinguish the syntactical rules for condition expressions and reactive expressions (as defined in the subsequent paragraphs), the rules of the common Boolean expressions have to be replicated for both expressions such that each occurrence of BE is substituted by CE for the condition expression or by RE for the reactive expression.

Condition Expression

A *condition expression* is a specialisation of a Boolean expression that refers (via its atomic elements) to either states of Behavior instances or to Condition statements (that are defined locally or by other instances). A *condition reference* thus consists of an optional reference to an instance and a reference identifier to the addressed Condition or State label. To illustrate which kinds of atomic elements have to be considered for a condition expressions, two statements of the running examples introduced in Chapter 1 are cited:

Firstly, the MS example specifies a statement which contains a condition expression notifying the system whether the second redundancy structure is no more fulfilled:

```
not C.RS1 & not (2 oo {S1.good, S2.good, S3.good, S4.good})
  guards C.⟨rs2f⟩
```

Hereby, references to all four sensors, encapsulated by an operator which states that at least two out of all four sensors have to be in good shape, assert on the question whether the second phase will also have failed. The reference rs2f triggers a guard/forward label in case that the answer to that question will be positive.

Secondly, a Condition statement can be found in the FTN example. It consists of an expression which conjuncts all link states l[i] arising iteratively by the expandable operator expression:

```
Condition nfailed = AND(i in {1 .. numLink}) l[i].cFailed
```

These two examples require defining a number of atomic elements for a condition expression:

- If a condition reference $(i,c) \in \mathfrak{Ref}_C = \mathfrak{Ref}$ refers to a local condition x, the optional reference i will remain empty, whereas the reference label c will be set to x. If it refers to a remote forward or to a state inside a behaviour instance, its optional reference i will also be set to the referred instance identifier. The corresponding grammar rule is given by:

```
RefCond: (i=ID '.')? c=ID;
```

- Also n-ary & , | and oo operators can be used, where sets of condition references are considered. Similar to expand statements in which similar repetitive expressions can be abbreviated, the iterator expressions can be used to operate on a body, i.e. a given set of reference operands. Therefore, $\mathfrak{RefSet}_C = \mathit{Opt}(\mathfrak{IE}) \times \mathcal{P}(\mathfrak{Ref}_C)$ defines the abstract representations of expandable expressions. Its concrete syntax rule is

```
RefCondSet:
  (ie=IE)? ( R+=RefCond | '{' R+=RefCond (',' R+=RefCond)*'}' );
```

  These atomic (n-ary) operator expressions are denoted as

$$A_C^{op} = \underbrace{\{\&, |\} \times \mathfrak{RefSet}_C}_{\text{for } \& \text{ and } |} \cup \underbrace{\mathbb{N} \times \mathfrak{RefSet}_C}_{\text{for oo}}.$$

  and their concrete syntax is given as follows

```
RefCondAnd: 'AND' R=RefCondSet
RefCondOr: 'OR' R=RefCondSet
RefCondOO: x=AE 'oo' R=RefCondSet
```

This implies that $A_C = A_C^{op} \cup \mathfrak{Ref}_C$ captures atomic elements of condition expressions. The associated syntactical rule is given as follows:

```
CE_atom: (RefCond | RefCondOr | RefCondAnd | RefCondOO);
```

Finally, a condition expression is defined as a specialisation of a Boolean expression comprising the specific atomic element given by A_C:

$$\mathfrak{CE} = \mathfrak{BE}^{A_C}$$

Reactive Expression

A *reactive expression* is a specialisation of a Boolean expression which (by its atomic elements) either directly refers to a transition guard label or to a forward label that is either locally defined or in another instance. For this purpose, *guard/forward label references* are defined which consist of an optional reference to an instance and a reference identifier to the addressed forward or transition guard label. In addition, sophisticated operators can be defined for atomic reactive expressions, as illustrated by the example of a guards statement taken from the MS model of Section 1.3:

```
C.Fail guards sync {
  C.⟨rep⟩, rm.⟨rep⟩, maxsync{S1.⟨repI⟩, S2.⟨repS⟩, S3.⟨repI⟩, S4
      .⟨repS⟩}
}
```

It makes use of the operators sync and maxsync. Therein not contained is the additional operator choose which might also be used when specifying an atomic reactive expression. Obviously, the operator expressions can be nested.

As a consequence, several kinds of atomic elements for a reactive expression are defined:

- Hereby a simple guard/forward label reference (i, l, dt) may refer to a local forward via the attribute l. The optional reference i will then remain empty. In case that the reference addresses a forward statement of a Module instance or a guard label of a Behavior instance, its optional reference i will be set to the referred instance identifier. Additionally, an optional distribution type dt is designated which is later used to determine the common distribution type between cooperating transitions. Accordingly, references are defined as follows:

  $$(i, l, dt) \in \mathfrak{Ref}_R = \mathcal{NS} \times \mathfrak{ID} \times Opt(D_{types})$$

  The corresponding grammar rule is given by:

  ```
  RefR: (i+=ID '.')? '<'l=ID'>';
  ```

- When using operators of the set $\mathcal{OP} = \{sync, maxsync, choose\}$, atomic (reactive) operator expressions or simple guard/forward references are required to be recursively embedded. Therefore, the universal set of atomic elements A_R for reactive expressions is given by

  $$A_R := \mathfrak{Ref}_R \cup A_R^{op},$$

  where the operator expressions are recursively defined following

  $$A_R^{op} := \mathcal{OP} \times Opt(\mathfrak{IE}) \times \mathcal{P}(A_R)$$

A tuple $(op, i, A) \in A_R^{op}$ hereby denotes the operator op, the optional iterator expression i and the set of atomic operands A.

The corresponding syntax rules are given as follows:

```
RefRSync: 'sync' R=RefReactSet;
RefRChoose: 'choose' R=RefReactSet;
RefRMaxSync: 'maxsync' R=RefReactSet;
```

They can be embedded as described by the following rule which also allows modellers to apply an iterator expression in order to expand the contained body:

```
RefReactSet:
  (ie=IE)? (R+=RE_atom | "{" R+=RE_atom ("," R+=RE_atom)+ "}");
```

The concrete grammar rule to construct an atomic reactive expression (corresponding to the abstract representation of A_R) is given by

```
RE_atom: (RefR | RefRSync | RefRChoose | RefRMaxSync);
```

A reactive expression is hence a specialisation of a Boolean expression that contains the specific atomic elements given by A_R:

$$\mathfrak{RE} = \mathfrak{BE}^{A_R}$$

The parsing rules initially restrict the definition of reactive expressions $\mathfrak{RE}$ to atomic elements $A_R \subset \mathfrak{RE}$. An atomic element of a reactive expression may thus refer to either a guard label or a forward label, or it may represent an embedding of reactive operator expressions sync, maxsync and choose.

As reactive expressions may be defined in choice, the Boolean operators $\&$, $|$, $\neg$ cannot be applied directly. This allows preserving the algebraic structure (when the transformation takes place) despite allowing choices. Only after resolving the reactive expressions, as given in the transformation defined in Section 3.3.3, the reactive operator expressions will be transformed into their corresponding Boolean expression representation.

```
forward ⟨x⟩ to {
   ⟨a⟩
   ⟨b⟩
}
```

```
forward ⟨y⟩ to {
   ⟨a⟩
   ⟨b⟩
}
```

Figure 2.6: Equivalent forward statements <x> and <y>

In order to demonstrate the above-mentioned course of action, a concrete example will be sketched: The result of resolving `sync{<x>,<x>}` will be equivalent to `sync{<x>,<y>}` if the forwards defining the references `<x>` and `<y>` are equivalent with regard to their content as given in Figure 2.6. When constructing `<x>&<x>` by parsing `sync{<x>,<x>}`, a substitution axiom can be applied which reduces `<x>&<x>` to `<x>`, whereas for `<x>&<y>` no substitution can be applied. Nevertheless, the outcome should be equivalent as the forward of both is equivalent. This is not the case here, as `<x>` is substituted with the choices `<a>` and `<b>`, whereas `<x>&<y>` is substituted with `<x>&<x>`, `<x>&<y>`, `<y>&<x>` and `<y>&<y>`. To preserve homomorphism for the reactive expressions used by guards and forward statements, its operators may not directly be transformed into the Boolean expression notation as long as its contained elements are not resolved.

Chapter 3

Transformation Semantics

The preceding chapter defined the abstract and concrete syntax of LARES. Based on the abstract syntax, this chapter formally defines transformations which, if applied to a LARES model, will yield a model of the targeted formalism. The underlying formal execution semantics of the target formalism is accordingly used to define the execution semantics of LARES in this chapter. Hereby, Section 3.1 defines abstract functions which denote how a given LARES model is traversed. Section 3.2 details how the visibility of definitions is queried and updated. The subsequent three sections deal with the transformations that construct the desired target models from a LARES model as depicted by Figure 3.1. Section 3.3 deals with *in-model* transformations by which the model instantiation is performed and all references are resolved. Their execution yields an expanded LARES model called LARES$_{\text{BASE}}$.

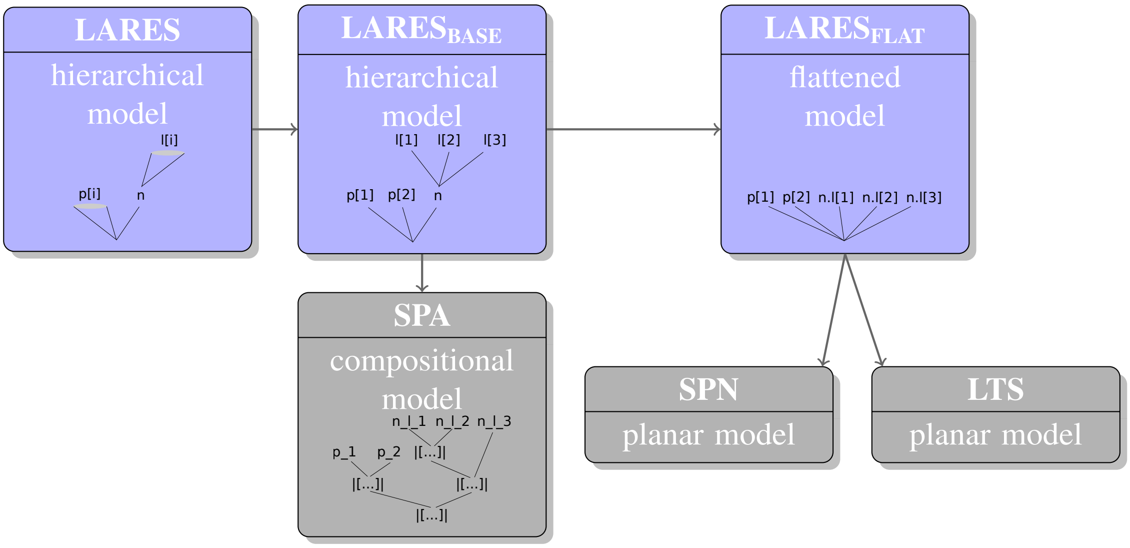


Figure 3.1: LARES transformations aim at either hierarchic/compositional or planar models

For this purpose, Subsection 3.3.1 details the resolution of parameters in order to construct the instance tree, followed by Subsection 3.3.2 which formalises the resolution of generative expressions for State and Condition statements, whereas Subsection 3.3.3 formalises the resolution of reactive expressions referring to forward statements. The stepwise application of the above transformations yield a LARES$_{\text{BASE}}$ model. On the basis of this model, the LARES approach aims at addressing different solver languages. The two subsequent sections are dedicated to the transformations into the target formalisms. These target formalisms can be classified into hierarchical and planar formalisms. If it is required to preserve the hierarchical structure of a LARES model, LARES$_{\text{BASE}}$ needs to be the starting point. Otherwise, if planar formalisms such as Petri nets or labelled transition systems (LTS) (which can be seen as Markov chains if transition rates are provided) are targeted, an intermediate transformation will have to be performed which flattens the instance tree. To give LARES a sound semantics, two transformations are presented, where each one maps to a distinct target formalism:

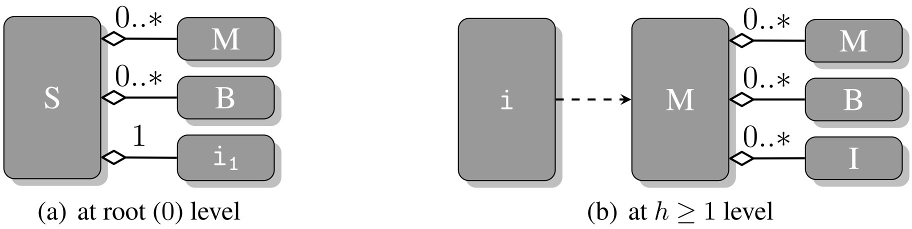
- Section 3.4 specifies a transformation into LARES$_{\text{FLAT}}$, a hierarchy-resolved variant of LARES$_{\text{BASE}}$, and defines how reachability is performed on a LARES$_{\text{FLAT}}$ model in order to construct the corresponding LTS.
- Section 3.5 specifies the transformation from LARES$_{\text{BASE}}$ into the stochastic process algebra language of the CASPA tool [11].

Each transformation defines a semantics for LARES, expressed by means of the addressed target formalism. It is required that behaviourally equivalent models are constructed by both transformations. Since the CASPA SPA semantics is given by means of LTS, a therewith constructed model should be equivalent to the result obtained by performing the reachability as described in Section 3.4. For this reason, Section 3.6 details at the end of this chapter how the definitions of two diverse transformations can be used to validate the intended LARES semantics. The correctness of the implementation is assured by cross-checking the equivalence of the resulting state space (which is obtained by the application of both transformations). This also enables the developer to preserve the implemented semantics in spite of code alteration. Furthermore, a formal proof is sketched in order to show the equivalence of both transformation semantics.

Apart from the transformations into the target languages formalised in Section 3.4 and Section 3.5 also Petri net formalisms (e.g. TimeNet4 [141] and SPNP [42]) have been addressed. However, both transformations are currently not further pursued and are therefore not discussed in this work.

3.1 Traversing LARES Models

A general definition is given firstly on how to traverse a model throughout its tree structure before formally denoting the transformations that, if applied in sequence, will yield the corresponding LARES$_{\text{BASE}}$ model. As depicted in Figure 3.2(a), the root node S of a LARES specification contains a System definition $\mathtt{i}_1$ as a specific Module instance serving as a singleton which may contain several definitions of Modules and Behaviors. Each Module instance statement i refers to its assigned Module definition which in turn may contain further Module and Behavior definitions and Module instances. This recursive structure is illustrated by Figure 3.2(b).



(a) at root (0) level

(b) at $h \geq 1$ level

Figure 3.2: Recursive structure of a LARES model

Ensuing from LARES' recursive structure, an abstract type of traversal function is defined. It is capable of propagating pieces of information in the direction to its leaves (subsequently denoted as *forward propagation*). It performs certain computations on each level using the local and the propagated information and aggregates the results of substructure transformations along the paths in the direction to its root node (*backward propagation*)

Traversal Function

The LARES abstract traversal function iterates over the instance tree of a LARES model. It is used to transform the instance tree by successively applying a forward and a backward function. Several transformations are defined, each one is realised by a traversal function τ which is parametrised by a forward propagation function ζ and a backward propagation function β. The parametrised traversal function $\tau_{\zeta,\beta}$ is thus declared as

$$\tau_{\zeta,\beta} : \mathfrak{I}_{\mathcal{M}} \times \mathfrak{FW} \to \mathfrak{BW}$$

Hereby $\mathfrak{I}_{\mathcal{M}}$ denotes an instance element of a LARES model, $\mathfrak{FW}$ represents the type of information which is propagated towards the leaf nodes of the traversed tree (*forward information*) and $\mathfrak{BW}$ represents the type of information which is computed taking the processed substructure of an element into account (*backward information*).

The abstract *forward function* ζ performs a computation which uses the current Module instance and the incoming forward information in order to return a tuple consisting of the (possibly) preprocessed Module instance and forward information:

$$\zeta : \mathcal{I}_{\mathcal{M}} \times \mathfrak{FW} \to \mathcal{I}_{\mathcal{M}} \times \mathfrak{FW}$$

The abstract *backward function* β depends on $\mathcal{I}_{\mathcal{M}} \times \mathfrak{FW}$ returned by ζ and the set of pieces of backward information $\mathcal{P}(\mathfrak{BW})$ obtained by the recursive transformation $\tau_{\zeta,\beta}$ of substructure instances. It is responsible for the determination of the backward information associated with the considered Module instance. Its definition is as follows:

$$\beta : \mathcal{I}_{\mathcal{M}} \times \mathfrak{FW} \times \mathcal{P}(\mathfrak{BW}) \to \mathfrak{BW}$$

Based on the function definitions of ζ and β, a transformations is realised by $\tau_{\zeta,\beta}$ which is formally defined as a recursive function which traverses a LARES model structure (as introduced in Section 2.1):

$$\tau_{\zeta,\beta} : (i, fw) \mapsto \beta(i', fw', \{\tau_{\zeta,\beta}(c_i, fw') \mid c_i \in i'.t.b.I_M\}) \text{ , where } (i', fw') := \zeta(i, fw)$$

Hereby, the notation $i'.t.b.I_M$ denotes the projection to the set of child instances. For each child instance c_i, a recursive descent is performed taking the obtained forward information fw' into account (cf. Figure 3.2(b)).

In order to reduce the given definition of $\tau_{\zeta,\beta}$ to its essentials, the following has to be executed: At each node, the obtained forward information is used and processed by a forward function ζ. The determined forward information is used to process all child instance substructures c_i by a recursive application of $\tau_{\zeta,\beta}$. The calculated results thereof are used to perform the final transformation of the given instance i.

Figure 3.3 illustrates the recursive transformation for a single Module instance nb, representing a northbridge of an imaginary mainboard model mb. The mainboard has furthermore a cpu with two caches c_1, c_2, a ram and a southbridge sb. Another simple transformation, which might serve as an entry-level example for the purpose of illustrating the use of the traversal function, is explained in Section 3.4.1. Hereby, the instance structure is flattened in contrast to the illustration given in Figure 3.3, which preserves the instance structure.

Before defining the concrete transformation steps, the *scoping semantics* is prefixed. Note that in case a function is not exhaustively defined for handling a given argument (for instance due to a reference to a non-existing entity), the model is considered $invalid$.

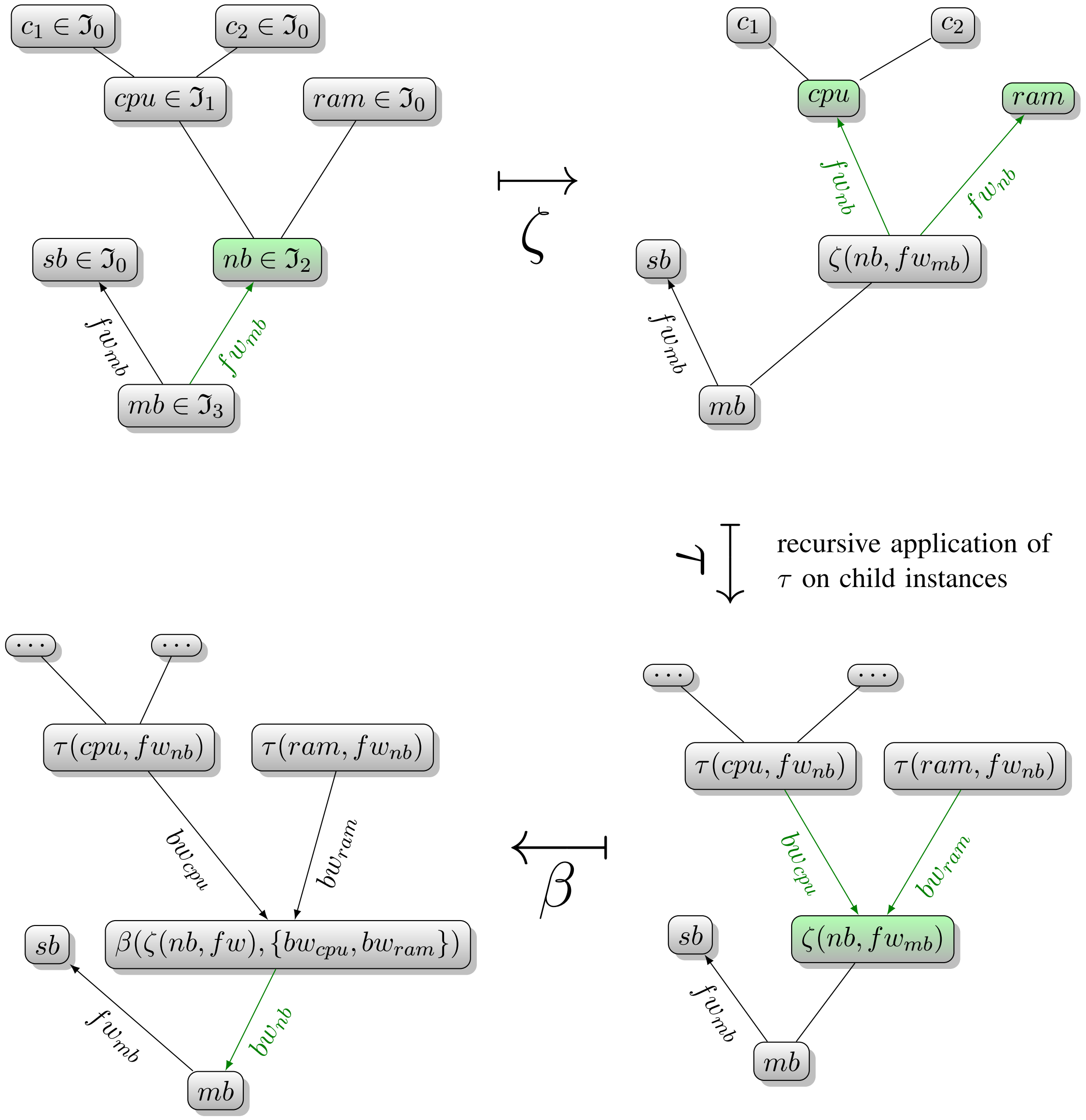


Figure 3.3: Illustration of a transformation step of the northbridge instance nb using the recursive traversal function τ. The function execution splits into a preprocessing step ζ (which calculates the forward information fw_{nb}), a recursive descent for each child element by τ and a post-processing step β (which ultimately calculates the backward information bw_{nb} of the transformation applied to nb).

3.2 Scoping Semantics

Each Module or Behavior definition specified in a LARES model has a scope. This means that a definition is only locally visible there, where it was defined, or in the substructures of the definition environment. Whenever there is an instantiation statement, the referred definition needs to be *in scope*, i.e. it must be visible at the location of its reference, otherwise the model is *invalid*.

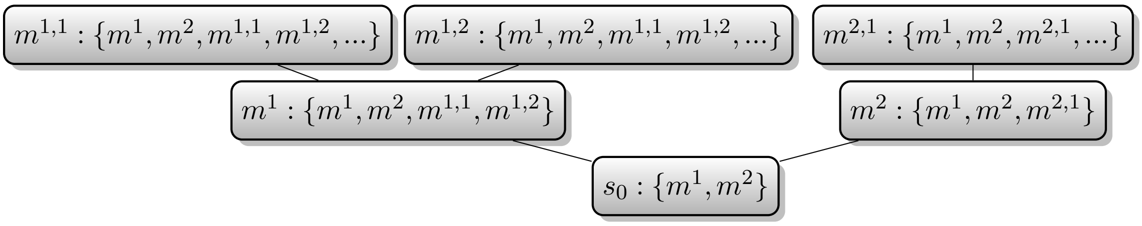

$m^{1,1} : \{m^1, m^2, m^{1,1}, m^{1,2}, ...\}$
$m^{1,2} : \{m^1, m^2, m^{1,1}, m^{1,2}, ...\}$
$m^{2,1} : \{m^1, m^2, m^{2,1}, ...\}$
$m^1 : \{m^1, m^2, m^{1,1}, m^{1,2}\}$
$m^2 : \{m^1, m^2, m^{2,1}\}$
$s_0 : \{m^1, m^2\}$

Figure 3.4: Scopes of Behavior or Module definitions: Visibility of abstract definitions (e.g. within m^1 the Module definitions $\{m^1, m^2, m^{1,1}, m^{1,2}\}$ are in scope and can be instantiated)

A trimmed snippet, taken from the FTN example presented in Section 1.2, is used in order to illustrate scopes of definitions.

```
Module Network(numLink) : R←Repair {
  Module Link : BLink {...}
  ...
}
...
System FTN : BSys {
  Instance n of Network(numLink=2)
  ...
}
```

In this snippet, the `Network` and the `FTN` definitions are stated in the same node. The `Network` definition is therefore visible in both definitions representing the subtrees of their common definition environment. As a consequence, the `Network` can be instantiated in the `FTN` definition. In contrast, the `Link` definition which is given inside the `Network` definition can only be instantiated within the `Network` definition and its associated substructures.

This can be represented by a tree of abstract definitions. Each definition is uniquely denoted by m^p, where p encodes the definition's location in the tree (regarding its path). In order to abstract from a specific path, the distance from the root node is given by the notation m_h which is implicitly derived from p by stating $m_{|p|}$. For each node inside the tree, a set of visible definitions is given to distinguish whether a definition is *in scope* and can therefore be used for an instantiation (cf. Figure 3.4).

The question whether a referred definition is *in scope* is answered by a function $inscope$ which takes an identifier Σ^+ denoting the definition to be instantiated and the set of visible definitions inside the current node. A function $inscope_M$ for Module and $inscope_B$ for Behavior definitions are hence needed to be distinguished:

$$inscope_M : \Sigma^+ \times \mathcal{P}(\mathcal{M}) \rightarrow \mathcal{M} \qquad inscope_B : \Sigma^+ \times \mathcal{P}(\mathcal{B}) \rightarrow \mathcal{B}$$

A LARES model will only be valid if all addressed definitions are *in scope*. According to this, the preceding function declarations are defined as

$$\begin{array}{llll} inscope_M: & (l, M) & \mapsto m & \text{if } \exists m \in M : m.l = l \text{ (else } invalid \text{ model), and} \\ inscope_B: & (l, B) & \mapsto b & \text{if } \exists b \in B : b.l = l \text{ (else } invalid \text{ model).} \end{array}$$

Inside each substructure further definitions may be specified which have the same name as another definition that is already *in scope*. In order to unambiguously resolve a reference to a definition, the property $\nexists m_1, m_2 \in M : m_1.l = m_2.l$ (where $m_1 \neq m_2$) and $\nexists b_1, b_2 \in B : b_1.l = b_2.l$ (where $b_1 \neq b_2$) has to hold for both M and B respectively. The property means that whenever new definitions are made and hence become part of the local scope information, equivalently named definitions have to be removed.

The $inscope$ functions are used in Section 3.3.1 in order to construct the instance tree of the model: As illustrated in Figure 3.2(a), at the root level (0) of a LARES specification $s_0 \in \mathcal{LARES}$ the sets of available Module and Behavior definitions are given by $B_0 = s_0.B$ and $M_0 := s_0.M$. The System instance $s_0.i_1$ is initially associated with its definition $s_0.i_1.t \in \mathcal{M}$. All along it is assumed that each definition stated inside a node (which might be a Module body or the root node of the LARES specification) has a unique name. Otherwise, the model is not well-defined (i.e. $invalid$). When processing arbitrary (sub)instances, the visible set of definitions of the predecessor node has to be merged with the local definitions with regard to the recursive scheme shown in Figure 3.2(b). Let the set of visible definitions for Module definitions of the predecessor be M_{h-1}. An update function Δ_M has to be applied in order to construct the set of visible definition M_h^i following

$$M_h^i = \Delta_M(M_{h-1}, i_h) \text{ for an instance } i_h \text{ at level } h.$$

Let the system instance i_1 be given (at root level). The update function $\Delta_M(M_0, i_1)$ will provide the set of visible definitions inside the System definition. The scope of Module definitions will be successively updated along an instance path. All updated sets of visible definitions are used to carry out the instantiation process (as detailed in Section 3.3.1) in order to enable the resolution of references to definitions and therefore, ultimately, to construct the instance tree.

The update function for the set of visible Module definitions is defined such that infinite recursive structures are allowed. This will by chance end-up in a non-terminating transformation if modellers do not pay attention. Each time an identifier of a definition $m_l \in M_{local}$ is equal to an identifier of a definition $m \in M_{h-1}$, m will be discarded in the constructed set of visible definitions M_h. The update function Δ_M is hence defined as

$$\Delta_M : (M_{h-1}, i_h) \mapsto M_{local} \cup \{\, m \in M_{h-1} \mid \nexists m_l \in M_{local} : m_l.l = m.l\}$$

where $m_{i_h} = inscope_M(i_h.l, B_{h-1})$ and $M_{local} = m_{i_h}.b.M$. In case it is required to disallow infinite recursive structures, an alternative definition may be used that restricts the scope of definitions by removing an instantiated Module definition m_{i_h} from the scope:

$$\Delta_M^{alternative} : (M_{h-1}, i_h) \mapsto M_{local} \cup \{\, m \in M_{h-1} \setminus \{m_{i_h}\} \mid \nexists m_l \in M_{local} : m_l.l = m.l\}$$

Correspondingly, an update for the set of visible Behavior definitions B_h^i at instance i_h at level h is performed by

$$B_h^i = \Delta_B(B_{h-1}, i_h)$$

where the update function Δ_B is defined in a manner similar to the definition of Δ_M. Since a Behavior definition does not allow any further instantiations, Δ_B can be defined as

$$\Delta_B : (B_{h-1}, i_h) \mapsto B_{local} \cup \{\, b \in B_{h-1} \mid \nexists b_l \in B_{local} : b_l.l = b.l\}$$

where $B_{local} = inscope_M(i_h.l, B_{h-1}).b.B$. The update functions Δ_M and Δ_B are used in Section 3.3.1 for determining the sets of visible definitions. This information will be propagated along the paths of the instance tree towards the leaf nodes.

3.3 Transformation into LARES$_{\text{BASE}}$

The transformation into a LARES$_{\text{BASE}}$ model (i.e. an instantiated reference-resolved representation of a LARES model) is subdivided into three consecutive steps. All steps could just as well have been defined within a single transformation step. For the sake of comprehensibility and maintainability they have been subdivided instead. The first step only considers the Instance and Initial statements. It performs the instantiation of addressed (visible) Behavior and Module definitions. A different instantiation parametrisation of Behavior and Module definitions leads to a parameter-dependent instance tree. The first transformation step is hence denoted as the *parameter expansion* transformation. For the subsequent transformation step all condition expressions as used in `Condition`, `forward` and `guards` statements are resolved. Due to the restriction of being solely allowed to refer to either local statements or statements specified in the children at adjacent level, a condition expression will be considered *unresolved* if it contains unresolved references referring to other Condition statements (apart from *resolved* references that refer directly to states of Behavior instances). The transformation into *resolved* condition expressions which then exclusively contain direct references to states of Behavior instances is called *condition expansion*. The final transformation step into the LARES$_{\text{BASE}}$ representation resolves *reactive* expressions. A *reactive* expression will be denoted *unresolved* if it refers to guard labels defined by other forward statements. A reactive expression will be regarded as being *resolved* if it only directly refers to guard labels of Behavior instances. Accordingly, the *forward expansion* transformation substitutes all guard label references, such that the model ends up with guards statements that solely contain *resolved* reactive expressions.

In the following paragraphs, these transformation steps are explained in detail by providing their formal semantics and are simultaneously illustrated with the help of running examples.

3.3.1 Parameter & Instance Tree Expansion

This section explains how to construct a specific instance tree of intermediate Module instance nodes and Behavior instance nodes (as leaves) using the instantiation parameters of the last recursion step and the addressed initial configuration. Beginning with an explanatory part, the formal definition of the transformation will be given subsequently.

When starting at the root node of a LARES model, all parameters that are defined for the system instance have to be identified first. In order to process the System instance definition, all expressions that depend on parameters have to be evaluated. Since System or Module definitions may also contain expand statements, the evaluation of the associated iterator expressions leads to a number of combinations of additional variable definitions (apart from the parameter variables) obtained by the cross product of their iterator expression evaluations. The resulting tuples can be regarded as sets of defined parameters which are used in addition to the existing ones to resolve the statements given inside the expand statement. As expand statements can be nested, this kind of process is recursive by nature.

The fault tolerant network example is now used as an illustrating example. In its System definition no parameters are defined. But still, new variables may arise. One example for such a case can be found in the FTN System definition:

```
expand (i in {1 .. 2}) { Instance p[i] of Processor}
```

The snippet contains an iterator expression which includes a set expression that (in this case) does not depend on parameters. This means that an evaluation ϕ(i in $\{1\ ..\ 2\}$) will lead to the following set of singlets $\{(1), (2)\}$. The contained statement `Instance p[i] of Processor` will be expanded such that for each singlet in the evaluation of i, the statement is modified such that i is replaced by the corresponding value in the singlet.

This process will generate the following statements substituting the original expand statement:

```
Instance p[1] of Processor
Instance p[2] of Processor
```

As a further example, a condition expression is shown, where an iterator expression is applied in order to specify n-ary operations:

```
Condition srs = OR[i in {1 .. 2}] p[i].pfailed | n.nfailed
```

The above example is accordingly transformed into the following statement:

```
Condition srs = ((p[1].pfailed | p[2].pfailed) | n.nfailed)
```

In the System definition, the Instance statement of the network and the Initial statement will stay untouched as they are not parameter-dependent and therefore do not constitute new parameters. Since there is nothing more to expand, the scope information will be updated including the definitions stated locally. When applying the parameter expansion to the Module body, a recursive descent is performed over the subinstances, which in turn applies this transformation to all module definitions that are referred by the instantiations. Subsequently, the network instance with the defined parameter `numLink=3` is considered. Inside the definition of the `Network` Module, an expand statement which depends on the parameter `numLink` can be found:

```
expand (i in {2 .. numLink}) {
  Instance l[i] initially iStandby of Link
  ...
}
```

When the parameter expansion finishes processing the internal representation of the above snippet, the following statements are obtained:

```
Instance l[2] initially iStandby of Link
Instance l[3] initially iStandby of Link
l[1].cFailed & l[2].cStandby guards l[2].⟨swactive⟩
(l[1].cFailed & l[2].cFailed) & l[3].cStandby guards
  l[3].⟨swactive⟩
```

Furthermore, the Condition statement which specifies when the network will have failed is also processed and transformed into

```
Condition nfailed = (l[1].failed & l[2].failed & l[3].failed)
```

The same scheme is performed on all instantiations (which, for the current example, leads to the instance tree depicted by Figure 3.5).

In the next section, the exact formal semantics of this transformation step will be described exhaustively. For that purpose, the *forward information* $\mathfrak{FW}$, the *forward function* ζ, the *backward information* $\mathfrak{BW}$ and the *backward function* β will be defined.

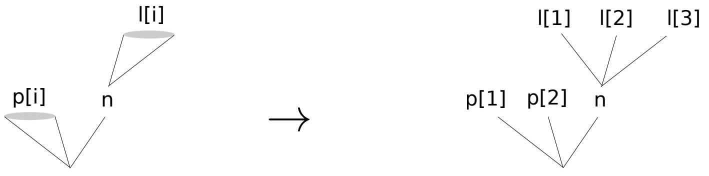


Figure 3.5: Constructed Instance Tree (of the FTN example) due to a given Parametrisation

The forward information is composed of the *scope information* representing visible Behavior and Module definitions as well as the initial configuration for each instance node:

$$\mathfrak{FW}_{pe} := \mathcal{P}(\mathcal{B}) \times \mathcal{P}(\mathcal{M}) \times \mathcal{P}(\mathfrak{Ref})$$

Hereby, the tuple $(B, M, IC) \in \mathfrak{FW}_{pe}$ denotes the set of visible Behavior definitions B, the set of visible Module definitions M and the set of references which address Initial statements. For the purpose of constructing an instance tree, the backward information is defined such that a Module instance will be obtained:

$$\mathfrak{BW}_{pe} := \mathcal{I}_{\mathcal{M}}$$

As indicated in Section 3.1, each concrete transformation performs specific computations, depending on different function definitions of ζ and β which are applied on each node of the structure traversed by some τ. The parameter expansion function is hence denoted as τ_{pe} using the information provided by $\mathfrak{FW}_{pe}$ in order to build up the instance tree from a given instance:

$$\underbrace{\tau_{\zeta_{pe},\beta_{pe}}}_{\tau_{pe}} : \mathcal{I}_{\mathcal{M}} \times \mathfrak{FW}_{pe} \rightarrow \mathfrak{BW}$$

Again, the FTN example snippet as given in Section 3.2 is used. Let the scope at the level of the specification node be given as defined in Section 3.2, where $B_0 := s_0.B$ and $M_0 := s_0.M$. The initial function call to perform the traversal and thereby to construct the instance tree is $\tau_{pe}(i_1, fw_1)$. The root instance i_1 represented by the tuple $(l, t, ())$ is implicitly derived from the System definition which itself is captured by t and excluded from the scope information which in turn is forwarded by fw_1 in order to ensure that the System definition can only be instantiated once:

$$fw_1 = (B_0, M_0, \emptyset)$$

Subsequently, the functions ζ_{pe} and β_{pe} are formally defined in order to denote the semantics of the parameter expansion transformation which constructs the model's instance tree.

Forward Propagation

The forward propagation function resolves all expressions that depend on parameters and updates the scope information in order to perform the instance tree construction.

Let the forward propagation function be defined such that the Module definition of the given instance is modified in accordance with the current parametrisation and such that the forward information fw containing visible definitions and the initial configuration are updated before the recursive descent takes place:

$$\zeta_{pe} : (i, fw) \mapsto (i', fw')$$

The above is responsible for evaluating and expanding the statements inside the Module body of i in order to further construct the instance tree. Therefore, a function b_{pe} is declared which performs evaluation and expansion regarding the given parametrisation:

$$b_{pe}^{Module} : \mathfrak{M}_{body} \times \mathfrak{PE} \rightarrow \mathfrak{M}_{body}$$

As shown for the initial step of the traversal, the reference to the Module definition of an instance i is resolved (which means that the instance henceforth contains the processed Module definition). The above parametrisation is determined by the defined parameters (i.e. parameters which have a defined value) within the Instance statement and, with a lesser priority, by the (default) parameters within the Module definition. In order to perform an instantiation, all parameters have to be defined in the end. Due to the twofold meaning of a System definition, being an abstract definition and an instance at once, all parameters have to be defined in the System's parameter expression.

For illustratory purposes, a Module definition of a `Network` defines a default value for the parameter `numLink`:

```
Module Network(numLink=3) {...}
```

It depends on the instantiation whether the default parameter value is overridden or not. The subsequent snippet firstly instantiates a network comprising two links, whereas the second instantiation uses the default value, leading to a network of three links:

```
Instance n_2links of Network(numLink=2)
Instance n_3links of Network
```

Generally speaking, the instance i, as a parameter of the *forward function*, is usually evaluated according to its parameter definitions. Except for the System instance, all instances are initially unresolved (which means that their reference to the abstract definition is not resolved yet). Let the forward parameter $fw \in \mathfrak{FW}_{pe}$ be represented as a tuple (B, M, IC).

The appropriate definition m is taken from the scope information in order to instantiate i (if the referred Module definition is not *in scope*, the model will be considered *invalid*), otherwise, if the processed instance is the System instance, the definition of the instance is already captured by $i.t$, which is handled here as a special case:

$$m = \begin{cases} inscope_M(i, M) & \text{if } i.t \in \mathfrak{Ref} \\ i.t & \text{else.} \end{cases}$$

The default parameters of the Module definition m and the evaluated parameters determined inside the Instance statement i are used to derive the actual parametrisation of the arising instance. A function is therefore declared which merges these two parameter expressions

$$p_{merge} : \mathfrak{PE} \times \mathfrak{PE} \to \mathfrak{PE}$$

such that the parameters of the first argument have priority over those of the second argument. Its definition is hence given by

$$\begin{aligned} & p_{merge} : (P_{prior}, P_{avail}) \mapsto \\ & \quad \{ p \in P_{prior} \mid \exists e \in P_{avail} : e.l = p.l \} \quad \cup \\ & \quad \{ p \in P_{avail} \mid \nexists e \in P_{prior} : e.l = p.l \} \end{aligned}$$

All parameters of P_{prior} (specified inside an Instance statement) whose names correspond to parameters defined in the Module definition P_{avail} become part of the actual parameter set. Furthermore, those parameters of the Module definition which are not defined by the instantiation are also included. Finally, the evaluation of the parameters is performed (following a topological order of the dependencies among the parameters).

Let $\phi^{\mathfrak{PE}} : \mathfrak{PE} \to \mathfrak{PE}$ declare an evaluation function. The actual parametrisation P can hence be determined:

$$P = \phi^{\mathfrak{PE}}(p_{merge}(i.t.p, m.p))$$

The property $\forall p \in P : \pi_2(p) \neq ()$ which requires all parameters of a Module definition to be defined has to be fulfilled by P, otherwise the model is invalid. As a result, the body can be resolved following $b = b_{pe}^{Module}(m.b, P)$. In order to derive the actual set of visible Module and Behavior definitions, the updated scopes $M' = \Delta_M(fw.M, i)$ and $B' = \Delta_B(fw.B, i)$ are determined. Furthermore, the set of delegates are partitioned into a set of Behavior instantiations and a set of Module instantiations:

$$d_B = \{ d \in m.d \mid \exists b \in B' : b.l = d.t.l \} \text{ and } d_M = d \setminus d_B$$

The body b of the Module instance is then transformed into b' such that the delegates which instantiate a Module definition are treated as additional Instances, i.e. $b'.I = b.I \cup d_M$.

Moreover, the initial configurations IC extracted from the forward information of the predecessor are used to determine the addressed configuration $ac \in IC : ac.i = i.l$ for the currently processed instance. A number of different cases for the definition of $init$ (i.e. the instantiations of the substructures) are captured. Firstly and having priority, the case in which an Initial statement of the current instance is referenced exactly once by the environmental Initial configuration is considered. Secondly, the case in which `initially` is used for instantiation by the environment is taken. The first Initial statement will be considered default if no external reference is given. Elsewise, neither will a reference be given nor can a default be chosen. For all other cases the whole model is considered invalid:

$$init = \begin{cases} ic.IC & \text{if } |\{ac\}| = 1 \text{ and } \exists ic \in b'.IC : ic.l = ac.l \\ ic.IC & \text{if } |\{ac\}| = \emptyset \text{ and } \exists ic \in b'.IC : ic.l = i.ic.l \\ \pi_1(b'.IC).IC & \text{if } \{ac\} = \emptyset \text{ and } b'.IC \neq \emptyset \\ \emptyset & \text{if } \{ac\} = \emptyset \text{ and } b'.IC = \emptyset \\ \emptyset & \text{else } \textit{invalid} \text{ model} \end{cases}$$

The initialisation references $init$ are subsequently used to resolve and instantiate the Behavior definitions originating from the delegate expression:

$$d_B^{inst} = \{ (inst^{Behavior}(d, inscope_B(d, B'), P, init.IC) \mid d \in d_B \},$$

For this purpose, the instantiation function $inst^{Behavior} : (d, m, P, IC) \mapsto (d.l, b, ic)$ is used. Hereby $d.l$ denotes the label of the delegate, b is a Behavior body, and ic represents the initial configuration. On the analogy of a Module instantiation, also for a Behavior instantiation the set of parameters $P = \phi^{\mathfrak{PC}}(p_{merge}(d.t.p, b.p))$ is determined in order to resolve the Behavior body $b = b_{pe}^{Behavior}(b.b, P)$ (the function $b_{pe}^{Behavior}$ body will be defined later). Additionally, the initial configuration $ic \in IC : ic.i = d.l$ is calculated in order to ultimately construct the Behavior instance $(d.l, b, ic) \in \mathfrak{I}_{\mathfrak{B}}$.

As a next step, the modified instance can be recomposed from its name, the resolved Module definition (comprising henceforth the determined parameters, the instantiated Behavior delegates and the resolved body) and an empty tuple (as the initial configuration already is part of the contained Behavior instances):

$$i' = (i.l, \underbrace{(m.l, P, d_B^{inst}, b')}_{\text{resolved Module definition}}, ())$$

The modified forward information fw' is ultimately given by $fw' = (B', M', init)$. Accordingly, the forward function can finally return the above results, i.e. $\zeta_{pe} : (i, fw) \mapsto (i', fw')$.

The Module body transformation function b_{pe} has to process each statement contained inside a Module's body. Let ${}^{v}\!/\!\phi$ denote a symbol for the substitution of variables by the values of the referred parameters and the subsequent evaluation of the dependent expressions. A number of functions are declared, each one being responsible for a set of specifically typed elements:

- ${}^{v}\!/\!\phi^{Instance} : \mathcal{P}(\mathcal{I}_{\mathcal{M}}) \times \mathfrak{PE} \to \mathcal{P}(\mathcal{I}_{\mathcal{M}})$
- ${}^{v}\!/\!\phi^{Condition} : \mathcal{P}(\mathcal{C}) \times \mathfrak{PE} \to \mathcal{P}(\mathcal{C})$
- ${}^{v}\!/\!\phi^{guards} : \mathcal{P}(\mathcal{G}) \times \mathfrak{PE} \to \mathcal{P}(\mathcal{G})$
- ${}^{v}\!/\!\phi^{forward} : \mathcal{P}(\mathcal{F}) \times \mathfrak{PE} \to \mathcal{P}(\mathcal{F})$
- ${}^{v}\!/\!\phi^{Initial} : \mathcal{P}(\mathfrak{IC}) \times \mathfrak{PE} \to \mathcal{P}(\mathfrak{IC})$
- ${}^{v}\!/\!\phi^{Probability} : \mathcal{P}(\mathfrak{Prob}) \times \mathfrak{PE} \to \mathcal{P}(\mathfrak{Prob})$

Apart from the above declarations, an additional function is declared which transforms the expand statements inside a Module's body:

$$\overset{Module}{{}^{v}\!/\!\phi^{expand}} : \mathcal{P}(\mathfrak{M}_{expand}) \times \mathfrak{PE} \to \mathcal{P}(\mathfrak{M}_{body})$$

The function b_{pe} which modifies and recomposes a Module body can now be defined using the preceding functions declarations:

$$b_{pe} : (b, P) \mapsto \begin{pmatrix} \emptyset \\ \emptyset \\ {}^{v}\!/\!\phi^{Instance}(b.I, P) \cup X_I \\ {}^{v}\!/\!\phi^{Condition}(b.C, P) \cup X_C \\ {}^{v}\!/\!\phi^{guards}(b.G, P) \cup X_G \\ {}^{v}\!/\!\phi^{forward}(b.F, P) \cup X_F \\ {}^{v}\!/\!\phi^{Initial}(b.IC, P) \cup X_{IC} \\ \emptyset \\ {}^{v}\!/\!\phi^{Probability}(b.P, P) \cup X_P \end{pmatrix} \quad \begin{array}{l} \text{where for each } \square \in \{I, C, G, F, IC, P\}, \\ X_{\square} := \bigcup \left\{ \widehat{b}.\square \mid \widehat{b} \in {}^{v}\!/\!\phi^{expand}_{Module}(b.E, P) \right\} \end{array} \tag{3.1}$$

Hereby, each subset of the partition, obtained by the transformation of all expand statements, contains only statements of a specific type. These statements are included in the new body, whereas the original expand statements are omitted.

The subsequent paragraphs define all parameter substitution and evaluation functions which are applied within the function definition (3.1). For this purpose, several function declarations are used dealing with the parameter substitution and evaluation of common sublanguage expressions:

- Set Expression Resolution:
  ${}^{v}\!/\!\phi^{\mathfrak{SE}} : \mathfrak{SE} \times \mathfrak{PE} \rightarrow \mathfrak{SE}$
- Condition Expression Resolution:
  ${}^{v}\!/\!\phi^{\mathfrak{CE}} : \mathfrak{CE} \times \mathfrak{PE} \rightarrow \mathfrak{CE}$
- Reactive Expression Resolution:
  ${}^{v}\!/\!\phi^{\mathfrak{RE}} : \mathfrak{RE} \times \mathfrak{PE} \rightarrow \mathfrak{RE}$
- Identifier Resolution:
  ${}^{v}\!/\!\phi^{\mathfrak{ID}} : \mathfrak{ID} \times \mathfrak{PE} \rightarrow \mathfrak{ID}$
- Reference Resolution:
  ${}^{v}\!/\!\phi^{\mathfrak{Ref}} : \mathfrak{Ref} \times \mathfrak{PE} \rightarrow \mathfrak{Ref}$
- Arithmetic Expression Resolution:
  ${}^{v}\!/\!\phi^{\mathfrak{AE}} : \mathfrak{AE} \times \mathfrak{PE} \rightarrow \mathfrak{AE}$
- Set Expression Evaluation:
  $\phi^{\mathfrak{SE}} : \mathfrak{SE} \rightarrow \mathcal{P}(\mathbb{R})$
- Parameter Expression Evaluation:
  $\phi^{\mathfrak{PE}} : \mathfrak{PE} \rightarrow \mathfrak{PE}$

Their definitions are deferred to the Appendix B for purposes of readability.

The instance substitution function ${}^{v}\!/\!\phi^{Instance}$ processes a given set of Instance statements, where each Instance subexpression, consisting of a label, a type reference and an initial configuration, is substituted by the parameter values corresponding to the contained variables:

$$ {}^{v}\!/\!\phi^{Instance}(I, P) \mapsto \{\, \underbrace{({}^{v}\!/\!\phi^{\mathfrak{ID}}(i.l, P), {}^{v}\!/\!\phi^{\mathfrak{Ref}}(i.t, P), {}^{v}\!/\!\phi^{\mathfrak{Ref}}(i.ic, P))}_{\hat{i} \in \mathfrak{I}} \mid i \in I \} $$

The conditions substitution function ${}^{v}\!/\!\phi^{Condition}$ processes a given set of Condition statements, in order to substitute all variables of the contained subexpressions of each statement by their corresponding parameter values:

$$ {}^{v}\!/\!\phi^{Condition}(C, P) \mapsto \{\, \underbrace{({}^{v}\!/\!\phi^{\mathfrak{ID}}(c.l, P), {}^{v}\!/\!\phi^{\mathfrak{CE}}(c.e, P))}_{\hat{c} \in \mathfrak{C}} \mid c \in C \} $$

The guards substitution function ${}^{v}\!/\!\phi^{guards}$ processes a given set of guards statement in order to obtain a set of substituted guards. The contained variables inside the elements of the guards' generative expression or *conditional reactive*s are substituted by the corresponding parameter values:

$$ {}^{v}\!/\!\phi^{guards}(G, P) \mapsto \{\, \underbrace{({}^{v}\!/\!\phi^{\mathfrak{CE}}(g.g, P), \hat{CR}, ())}_{\hat{g} \in \mathfrak{G}} \mid g \in G \}\text{ , where} $$

$$ \hat{CR} := \{\, ({}^{v}\!/\!\phi^{\mathfrak{CE}}(c, P), {}^{v}\!/\!\phi^{\mathfrak{RE}}(r, P)) \,|\, (c, r) \in g.CR \} $$

The forwards substitution function ${}^{v}\!/\!\phi^{forward}$ processes a given set of forward statements in order to obtain a set of substituted forwards. Hereby, each occurrence of a variable inside the label, the generative expression or the associated *conditional reactive*s are substituted by the corresponding parameter value:

$$
\begin{aligned}
{}^{\mathrm{v}}\!/\!\phi^{forward}(F,P) &\mapsto \{\, \underbrace{({}^{\mathrm{v}}\!/\!\phi^{\mathfrak{CE}}(f.c,P), {}^{\mathrm{v}}\!/\!\phi^{\mathfrak{ID}}(f.l,P), \hat{CR})}_{\hat{f}\in\mathfrak{F}} \mid f \in F\} \text{, where} \\
\hat{CR} &:= \{\, ({}^{\mathrm{v}}\!/\!\phi^{\mathfrak{CE}}(c,P), {}^{\mathrm{v}}\!/\!\phi^{\mathfrak{RE}}(r,P)) \mid (c,r) \in f.CR\}
\end{aligned}
$$

The initial substitution function ${}^{\mathrm{v}}\!/\!\phi^{Initial}(IC,P)$ is applied to the given Initial statements of the instantiated Module definition in order to substitute each occurrence of a variable by the value of the corresponding parameter:

$$
{}^{\mathrm{v}}\!/\!\phi^{Initial}(IC,P) \mapsto \{\, \underbrace{({}^{\mathrm{v}}\!/\!\phi^{\mathfrak{ID}}(ic.l), \{\, {}^{\mathrm{v}}\!/\!\phi^{\mathfrak{Ref}}(r) \mid r \in ic.IC\})}_{\hat{ic}\in\mathfrak{IC}} \mid ic \in IC\}
$$

The expand substitution function ${}^{\mathrm{v}}\!/\!\phi^{expand}_{Module}$ processes a set of expand statements, where the contained iterator expression is substituted and evaluated such that recursively all statements in the assigned body are resolved by the parameters obtained (hereby I., II., and III. distinguish three steps which will be detailed subsequently):

$$
{}^{\mathrm{v}}\!/\!\phi^{expand}_{Module}(E,P) \mapsto \{\, \underbrace{b_{pe}(e.b, p^{+}_{merge}(\hat{P},P))}_{\text{III.}} \quad \mid \quad \hat{P} \in \overbrace{\phi^{\mathfrak{IE}}(\underbrace{{}^{\mathrm{v}}\!/\!\phi^{\mathfrak{IE}}(e.ie,P)}_{\text{I.}})}^{\text{II.}} \wedge e \in E\,\}
$$

While p_{merge} allows only parameters which are also available in P_{avail}, p^{+}_{merge} includes all parameters from P_{prior} instead:

$$
p^{+}_{merge} : (P_{prior}, P_{avail}) \mapsto P_{prior} \cup \ \{p \in P_{avail} \mid \nexists e \in P_{prior} : e.l \overset{\wedge}{=} p.l\}
$$

I. Firstly, the function ${}^{\mathrm{v}}\!/\!\phi^{\mathfrak{IE}}$ processes all iterators of an iterator expression such that each occurrence of a variable corresponding to a parameter inside the iterator expression is substituted. To illustrate this, an example will be given:

$$
\textbf{expand} \ (\mathrm{i}\ \textbf{in}\ \{2\ ..\ \mathrm{numLink}\})\ \{\ \ldots\ \} \ \overset{{}^{\mathrm{v}}\!/\!\phi^{\mathfrak{IE}}}{\Longrightarrow} \ \textbf{expand} \ (\mathrm{i}\ \textbf{in}\ \{2\ ..\ 3\})\ \{\ \ldots\ \}
$$

Given an iterator expression $(l,s) \in \mathfrak{I}$, the function ${}^{\mathrm{v}}\!/\!\phi^{\mathfrak{SE}} : \mathfrak{SE} \times \mathfrak{PE} \to \mathfrak{SE}$ is used which substitutes the variables of the set expression matching the given parameters. The substitution function for an iterator expression can then be declared by ${}^{\mathrm{v}}\!/\!\phi^{\mathfrak{IE}} : \mathfrak{IE} \to \mathfrak{IE}$ and defined as

$$
{}^{\mathrm{v}}\!/\!\phi^{\mathfrak{IE}} : (I,P) \mapsto \{\, ({}^{\mathrm{v}}\!/\!\phi^{\mathfrak{ID}}(l,P), {}^{\mathrm{v}}\!/\!\phi^{\mathfrak{SE}}(s,P)) \mid (l,s) \in IE\}
$$

```
expand (j in {1 .. i}, i in {1 .. numLink}) { ... }
```

Figure 3.6: Example: Dependency among two iterators

II. Secondly, an evaluation $\phi^{\mathfrak{IE}}$ performed over all iterators of a single expand statements leads to a cross product. Each contained tuple constitutes a parameter set $\hat{P}$ which is further used by the recursive expansion process b_{pe}.

Note that specifications can occur which denote iterator expressions by which iterators depend on the evaluation of other iterators (as shown in Figure 3.6). For the purpose of handling these specifications, the evaluation has to be performed by abiding by a topological order based on the interdependencies of the iterators. In case of cyclic dependencies the whole model is considered $invalid$.

Let a function $ts : \mathfrak{IE} \to Seq(\mathfrak{I})$ be defined which determines a topological order among dependent iterators. A recursive definition of an evaluation function is subsequently given which constructs the cross-product by taking the interdependent parameters (arising stepwise during the evaluation) into account. The evaluation function of an iterator expression therefore sorts the iterators with regard to their dependencies and subsequently applies the step-wise cross product construction function $\phi^{ts:\mathfrak{IE}} : Seq(\mathfrak{I}) \times \mathfrak{PE} \to \mathcal{P}(\mathfrak{PE})$:

$$\phi^{\mathfrak{IE}} : ie \mapsto \phi^{ts:\mathfrak{IE}}(ts(ie), \emptyset)$$

As explained before, processing the iterators in a topological order of their interdependencies enables an iterator's evaluation in the absence of cyclic dependencies, so as to add a result of an intermediate processing step and reuse it in all subsequent steps in order to ultimately construct the cross product of evaluations:

$$\phi^{ts:\mathfrak{IE}} : (I, P) \mapsto \begin{cases} \{()\} & \text{if } I = () \\ \bigcup_{v \in \phi^{\mathfrak{SE}}(\mathbf{v}\not\phi^{\mathfrak{SE}}(s,P)))} \{p\} \times \phi^{ts:\mathfrak{IE}}(\pi_{tail}(I), P \cup \{p\}) & \text{else,} \end{cases}$$
where $(l, s) := \pi_{head}(I)$ and $p := (l, v)$

An examplary application of the iterator expression evaluation is shown in Figure 3.7.

III. The step-wisely constructed parameters $\hat{P}$ are merged with the parameters P brought in by the environment. The expand statement evaluation function can now process all contained statements and further embedded expand structures by applying b_{pe} on the body of the expand statement (as shown in the example given in Figure 3.8).

```
expand (j in {1 .. i},  i in {1 .. 2}) {
   ...
}
```

$$\overset{\phi^{\mathfrak{IE}}}{\Rightarrow}$$

```
expand ({
   (i=1, j=1),
   (i=2, j=1),
   (i=2, j=2)
}) {
   ...
}
```

Figure 3.7: Example: Evaluation of dependent iterators

```
expand ({
   (i=1, j=1),
   (i=2, j=1),
   (i=2, j=2)
}) { Condition c[i,j] = ... }
```

$$\overset{b_{pe}}{\Rightarrow}$$

```
Condition c[1, 1] = ...
Condition c[2, 1] = ...
Condition c[2, 2] = ...
```

Figure 3.8: Example: Iterative body expansion by evaluations arising from iterators

The Probability measure transformation function ${}^{v}\!/\!\phi^{Probability}$ processes a given set of Probability statements in order to substitute all variables of the contained subexpressions of each statement by their corresponding parameter values:

$$
{}^{v}\!/\!\phi^{Probability}(Pr, P) \mapsto \\
\bigcup_{pr \in Pr} \begin{cases} ({}^{v}\!/\!\phi^{\mathfrak{ID}}(pr.l, P), {}^{v}\!/\!\phi^{\mathfrak{CE}}(pr.c, P)) & \text{if } pr \in \mathfrak{Prob}_S \\ ({}^{v}\!/\!\phi^{\mathfrak{ID}}(pr.l, P), {}^{v}\!/\!\phi^{\mathfrak{CE}}(pr.c, P), {}^{v}\!/\!\phi^{\mathfrak{AE}}(pr.t, P)) & \text{if } pr \in \mathfrak{Prob}_T \end{cases}
$$

The Behavior body transformation function $b_{pe}^{Behavior}$ has to process each statement contained by a Behavior body. For this purpose, several functions are declared. Each one of these performs parameter substitution and evaluation for a specific type of element:

- ${}^{v}\!/\!\phi^{State} : \mathcal{P}(\mathcal{S}) \times \mathfrak{PE} \mapsto \mathcal{P}(\mathcal{S})$
- ${}^{v}\!/\!\phi^{Transitions} : \mathcal{P}(\mathcal{T}) \times \mathfrak{PE} \mapsto \mathcal{P}(\mathcal{T})$

Apart from these ones, an additional function is declared which transforms the expand statements inside a Behavior's body:

$$
{}^{v}\!/\!\phi^{Behavior}_{expand} : \mathcal{P}(\mathfrak{B}_{expand}) \times \mathfrak{PE} \mapsto \mathcal{P}(\mathfrak{B}_{body})
$$

Using the above declarations, the transformation b_{pe} of a Behavior can be defined as follows:

$$b_{pe} : (b, P) \mapsto \begin{pmatrix} {}^{\mathrm{v}}\!/\!\phi^{State}(b.S, P) \cup X_S \\ {}^{\mathrm{v}}\!/\!\phi^{Transitions}(b.T, P) \cup X_T \\ \emptyset \end{pmatrix} \quad \begin{array}{l} \text{where for each } \square \in \{S, T\}, \\ X_{\square} := \bigcup \left\{ \widehat{b}.\square \mid \widehat{b} \in {}^{\mathrm{v}}\!/\!\phi^{expand}_{Behavior}(b.E, P) \right\} \end{array}$$

As all declarations which are applied by the transformation of a Behavior body can be defined in analogy to those used for the transformation of a Module body, these definitions are omitted in this work.

Backward Aggregation

The backward aggregation function composes the Module instantiations of the currently processed instance. Therefore, the recursive application of the parameter expansion on each child instance of i returns the processed subinstance structures contained in BW. All of these structures are inserted into the body $b = m.b$ of i's Module definition $m = i.t$. This leads to a modified body b' which is used to construct the parameter resolved instance i'. By using the preceding notation, the *backward propagation* function is declared as $\beta_{pe} : \mathcal{I} \times \mathfrak{FW}_{pe} \times \mathcal{P}(\mathfrak{BW}_{pe}) \to \mathfrak{BW}_{pe}$ and defined as

$$\beta_{pe} : (i, fw, BW) \mapsto (i'),$$

where the new body $b' := (\emptyset, \emptyset, BW, b.C, b.G, b.F, \emptyset, \emptyset, b.P)$ is used to reconstruct the Module definition $m' := (m.l, \emptyset, m.d, b')$ in order to finally obtain the instance $i' := (i.l, m', ())$

3.3.2 Condition Expansion

The second transformation step processes all condition expressions appearing in Condition, forward and guards statements. Each condition expression may contain *unresolved* references to Condition statements specified locally or within Module subinstances. By resolving these references, a condition expression is expanded such that it only contains direct references to states within Behavior instances in order to be finally *resolved*. The whole recursive transformation is hence called *condition expansion*. As set out in the previous section, this section starts with an illustrative part and subsequently proceeds with the formal definition of the transformation.

Once the *parameter expansion* has constructed an instance tree of the model, as illustrated in Figure 3.9, all condition expressions need to be resolved recursively. Starting at the leaves of the Module instance tree, all condition expressions found are rebuilt such that

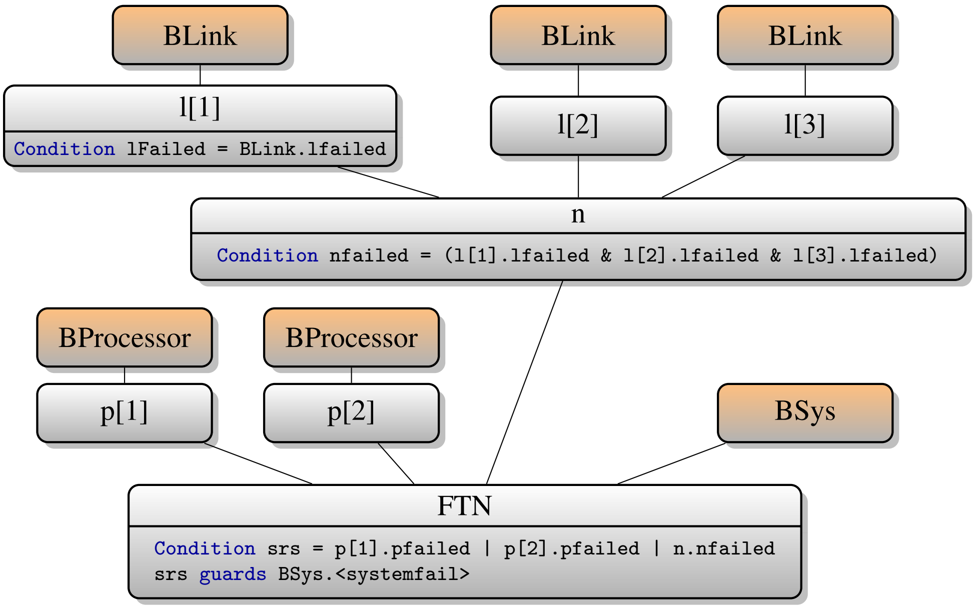

BLink
BLink
BLink
l[1]
Condition lFailed = BLink.lfailed
l[2]
l[3]
n
Condition nfailed = (l[1].lfailed & l[2].lfailed & l[3].lfailed)
BProcessor
BProcessor
p[1]
p[2]
BSys
FTN
Condition srs = p[1].pfailed | p[2].pfailed | n.nfailed
srs guards BSys.<systemfail>

Figure 3.9: FTN: The instance tree and Condition statements along a single path

there is no further dependency to other Condition statements. Since a leaf Module instance does not have further Module subinstances, they may only directly refer to state variables of their Behavior subinstances or to Condition statements specified locally.

The process of resolving dependencies is sensitive to the order of resolvable conditions, similar to the parameter dependencies arising through an iterator expression specification. As a consequence, the resolution process has to follow a topological order (in a manner similar to the way it was defined previously) which is based on dependencies to Condition statements. If a topological order can be determined, all dependencies (i.e. the referred Condition statements) for each unresolved condition expression will be resolved by the result of one of the preceding steps of the resolution process. This finally allows resolving the yet unresolved condition expression. The whole specification is invalid in case that no topological order for the local dependencies of a condition expression can be determined. The transformation which resolves the condition expressions of an instance is defined which obtains all *resolved* Condition statements of the Module subinstances by a preceding recursive descent. These are used to resolve the remote references (i.e. references that refer to statements of child instances) inside a condition expression by a substitution of the expressions inside the addressed Condition statements. Moreover, the local dependencies among the Condition statements have to be resolved in accordance with the thereby arising topological order to finally construct the *resolved* condition expressions of the given instance.

In Figure 3.9 the FTN example is stressed in order to illustrate the condition expansion by partially depicting the parameter resolved instance tree together with statements that include condition expressions along a single path. E.g. the network instance `n` includes a number of statements containing condition expressions:

```
(l[1].lfailed & l[2].lstandby) guards l[2].⟨swactive⟩

((l[1].lfailed & l[2].lfailed) & l[3].lstandby)
  guards l[3].⟨swactive⟩

Condition nfailed = (l[1].lfailed & l[2].lfailed & l[3].failed)
```

The first two condition expressions are part of a guards statement, whereas the third one belongs to a Condition statement. All contained condition expressions refer to Condition statements inside the associated Module definitions of the link subinstances `l[1]`, `l[2]` and `l[3]`.

The condition expansion is responsible for resolving each condition expression at an arbitrary level: From the point of view of a Module instance, all Condition statements inside the subinstances have already been resolved by a preceding recursive step. Thus, all local condition expressions can be modified such that references to Condition statements of subinstances are substituted by the corresponding resolved condition expressions. As a result, the substituted condition expressions only refer to state variables of Behavior instances and are hence *resolved*:

```
(l[1].BLink.lfailed & l[2].BLink.lstandby) guards l[2].⟨swactive⟩

((l[1].BLink.lfailed &  l[2].BLink.lfailed) & l[3].BLink.
    lstandby)
  guards l[3].⟨swactive⟩

Condition nfailed =
  (l[1].BLink.lfailed & l[2].BLink.lfailed & l[3].BLink.lfailed)
```

Being the last of steps in the recursive process, the statements of the system instance FTN containing condition expressions are transformed. This results in a guards statement containing only a resolved condition expression which itself represents the redundancy structure of the system:

```
((p[1].BProcessor.pfailed|p[2].BProcessor.pfailed) |
  ( n.l[1].BLink.lfailed &
    n.l[2].BLink.lfailed &
    n.l[3].BLink.lfailed )) guards BSys.⟨systemfail⟩.
```

Note that the FTN example does not comprise local dependencies between Condition statements, therefore no topological order has to be considered.

In the subsequent part, the introduced principle is completely formalised, including the local dependencies that were omitted in the example for the sake of simplicity: As it was described earlier, no information has to be forwarded in order to perform the condition expansion. For this reason, the forward information may only contain the empty set, i.e. $\mathfrak{FW}_{ce} = \{\emptyset\}$. In a way similar to the previous section, the information relayed backwards is always a Module instance and hence formally defined as $\mathfrak{BW}_{ce} = \mathfrak{I}$.

Now, the traversal function τ_{ce} is declared carrying out the condition expansion:

$$\tau_{ce} : \mathfrak{I}_{\mathcal{M}} \times \mathfrak{FW}_{ce} \to \mathfrak{BW}_{ce}$$

The forward propagation ζ_{ce} and the backward propagation β_{ce} remain to be defined.

Forward Propagation

As the expansion of condition expressions only depends on conditions (that can be found locally or inside substructures), the forward function ζ_{ce} is defined as the identity function:

$$\zeta_{ce} : (i, fw) \mapsto (i, fw)$$

Backward Aggregation

Condition expressions inside a given instance i are resolved by using the backward information $BW \in \mathfrak{BW}_{ce}$ obtained by a recursive descent of the condition expansion transformation of its subinstances. The arising instance i' containing solely resolved condition expressions is used as the information which is relayed backwards. The corresponding backward function is therefore defined as

$$\beta_{ce} : (i, fw, BW) \mapsto (i').$$

The reconstructed instance i' only contains condition expressions that were resolved by the addressed resolved Condition statements inside the processed subinstances. Therefore, the Module definition $m = i.t$ of the original instance i is processed. Hereby, the condition expressions inside the body $b = m.b$ are resolved by a function b_{ce} which employs the processed subinstances obtained via the backward information BW:

$$i' := (i.l, \underbrace{(m.l, \emptyset, m.d, b_{ce}(b, BW))}_{\in \mathcal{M}}, ()), \text{ where } m := i.t$$

Let ${}^{c}\!/\!\phi^{Condition}_{remote} : \mathcal{P}(\mathcal{C}) \times \mathcal{P}(\mathcal{I}) \rightarrow \mathcal{P}(\mathcal{C})$ substitute the addressed remote Condition statements of the subinstances which have already been resolved by the preceding recursive condition expansion step. Moreover, let $ts : \mathcal{P}(\mathcal{C}) \rightarrow Seq(\mathcal{C})$ be a function that establishes a topological order between the given (local) Condition statements based on their interdependencies. A further (recursive) function is declared which resolves these interdependencies by performing the condition expression substitution. Its first argument represents the (locally yet) unresolved Condition statements, whereas the second argument captures the (already) resolved (local) Condition statements:

$$ {}^{c}\!/\!\phi^{Condition}_{local} : Seq(\mathcal{C}) \times \mathcal{P}(\mathcal{C})) \rightarrow \mathcal{P}(\mathcal{C}) $$

In order to determine the resolved Condition statements C_{res} from the given set of local Condition statements $b.C$, the above functions can now be used:

$$ C_{res} = {}^{c}\!/\!\phi^{Condition}_{local}((ts \circ {}^{c}\!/\!\phi^{Condition}_{remote})(b.C, BW), \emptyset) $$

Hereby, the Condition statements inside the subinstances are substituted first. Then, by topological sorting, an order is established which allows substituting the addressed local Condition statements.

In addition, let ${}^{c}\!/\!\phi^{guards} : \mathcal{P}(\mathcal{G}) \times \mathcal{P}(\mathcal{C}) \times \mathcal{P}(\mathcal{I}_{\mathcal{M}}) \rightarrow \mathcal{P}(\mathcal{G})$ resolve the local guards statements, let ${}^{c}\!/\!\phi^{forward} : \mathcal{P}(\mathcal{G}) \times \mathcal{P}(\mathcal{C}) \times \mathcal{P}(\mathcal{I}_{\mathcal{M}}) \rightarrow \mathcal{P}(\mathcal{G})$ resolve the local guards statements and let ${}^{c}\!/\!\phi^{Probability} : \mathcal{P}(\mathfrak{Prob}) \times \mathcal{P}(\mathcal{C}) \times \mathcal{P}(\mathcal{I}_{\mathcal{M}}) \rightarrow \mathcal{P}(\mathfrak{Prob})$ resolve the local Probability statements. All these functions consider the resolved local Condition statements and the resolved remote Condition statements inside the processed set of subinstances.

With the above being given, the body substitution function $b_{ce} : \mathfrak{M}_{body} \times \mathcal{P}(\mathcal{I}) \rightarrow \mathfrak{M}_{body}$ can then be defined as

$$ b_{ce} : (b, I) \mapsto \begin{pmatrix} \emptyset \\ \emptyset \\ I \\ C_{res} \\ {}^{c}\!/\!\phi^{guards}(b.G, C_{res}, I) \\ {}^{c}\!/\!\phi^{forward}(b.F, C_{res}, I) \\ b.ic \\ \emptyset \\ {}^{c}\!/\!\phi^{Probability}(b.P, C_{res}, I) \end{pmatrix} $$

In the subsequent part of this section, the above declared functions are formally defined. The function resolves the remote dependencies ${}^{c}\!/\!\phi^{\mathit{Condition}}_{\mathit{remote}}$ is defined as

$$ {}^{c}\!/\!\phi^{\mathit{Condition}}_{\mathit{remote}} : (C, I) \mapsto \{\, (l, {}^{c}\!/\!\phi_{r}(c, I)) \mid (l, c) \in C \} $$

Hereby, the function ${}^{c}\!/\!\phi_{r} : \mathfrak{CE} \times \mathcal{P}(\mathcal{I}) \rightarrow \mathfrak{CE}$ (defined later in this section) is responsible for the resolution of a single condition expression inside a Condition statement by the resolved Condition statements of its subinstances.

The LARES syntax allows denoting local references such that Condition statements might induce cyclic dependencies. These are determined by a topological sorting algorithm, i.e if no topological order exists, the whole model is *invalid*. Otherwise, the resulting topological order will ensure that the dependencies can be resolved (such that the referred Condition statements are always processed before being addressed by a dependent statement). As indicated, the set of (already) resolved local Conditions has to be updated in each step. To illustrate the above, let the following statements be given inside an example model and let the addressed remote references, such as `B.c`), already be *resolved*:

```
Condition a = b & B.c
Condition b = C.c | D.c
```

As Condition `a` is dependent on Condition `b`, the order of their resolution process is crucial. The correct sequence will be established by performing the resolution process in the topological order (i.e. the Condition statements that are locally independent will be processed initially):

```
Condition b = C.c | D.c
Condition a = b & B.c
```

Accordingly, the Condition statement `b` has no local dependencies and can subsequently be used as a *locally resolved* statement which can be further used in order to resolve the yet unresolved local Condition statements.

The local substitution is recursively defined as

$$ {}^{c}\!/\!\phi^{\mathit{Condition}}_{\mathit{local}} : (C, C_{\mathit{res}}) \mapsto \begin{cases} C_{\mathit{res}} & \text{if } C = () \\ {}^{c}\!/\!\phi^{\mathit{Condition}}_{\mathit{local}}(C_{\mathit{tail}}, \{c'_{\mathit{head}}\} \cup C_{\mathit{res}}) & \text{else,} \end{cases} $$

The argument C can be split into the first Condition statement c_{head} and the remaining ones C_{tail}, i.e. $(c_{\mathit{head}}) \circ C_{\mathit{tail}} = C$. The Condition statement c_{head} is processed using a function ${}^{c}\!/\!\phi_{l} : \mathfrak{CE} \times \mathcal{P}(\mathcal{C}) \rightarrow \mathfrak{CE}$ which substitutes the addressed (resolved) local Conditions within an unresolved Condition expression. The locally resolved Condition statement is determined by $c'_{\mathit{head}} = (c_{\mathit{head}}.l, {}^{c}\!/\!\phi_{l}(c_{\mathit{head}}.c, C_{\mathit{res}}))$ and is included in the second argument of the next recursive call.

Note that ${}^{c}\!/\!\phi_l$ is a substitution function of condition expressions which will be defined soon.

Let ${}^{c}\!/\!\phi(ce, C, I) \mapsto {}^{c}\!/\!\phi_l({}^{c}\!/\!\phi_r(ce, I), C)$ define the Condition substitution function. The substitution function for guards statements is then defined as

$$
{}^{c}\!/\!\phi^{guards} : G, C, I \mapsto \{\, ({}^{c}\!/\!\phi(g.g, C, I), \hat{CR}) \mid g \in G\}, \text{ where}
$$

$$
\hat{CR} := \{\, ({}^{c}\!/\!\phi(c, C, I), r) \mid (c, r) \in g.CR\}
$$

Furthermore, the substitution function for forward statements is similarly defined by

$$
{}^{c}\!/\!\phi^{forward} : F, C, I \mapsto \{\, ({}^{c}\!/\!\phi(f.c, C, I), f.l, \hat{CR}) \mid f \in F\}, \text{ where}
$$

$$
\hat{CR} := \{\, ({}^{c}\!/\!\phi(e, C, I), re) \mid (e, re) \in f.CE\}
$$

Finally, two substitution functions ${}^{c}\!/\!\phi_l$ and ${}^{c}\!/\!\phi_r$ are defined considering local and remote resolved Condition statements, respectively:

$$
{}^{c}\!/\!\phi_l : (c, C_l) \mapsto \begin{cases} ({}^{c}\!/\!\phi_l(l, I), op, {}^{c}\!/\!\phi_l(r, I)) & \text{if } (l, op, r) = c \\ (\neg, {}^{c}\!/\!\phi_l(c', I)) & \text{if } (\neg, c') = c \\ c_l.c & \text{if } ((), l) = c \text{ and } \exists c_l \in C_l : c_l.l \triangleq l \\ c & \text{else} \end{cases}
$$

$$
{}^{c}\!/\!\phi_r : (c, I) \mapsto \begin{cases} ({}^{c}\!/\!\phi_r(l, I), op, {}^{c}\!/\!\phi_r(r, I)) & \text{if } (l, op, r) = c \\ (\neg, {}^{c}\!/\!\phi_r(c', I)) & \text{if } (\neg, c') = c \\ ns^{\mathfrak{CE}}(i.l, c_r.c) & \text{if } (r, l) = c \text{ and } \exists i \in I : i.l \triangleq r \text{ and} \\ & \quad \exists c_r \in i.t.b.C : c_r.l \triangleq l \\ c & \text{else} \end{cases}
$$

Note that for reasons of brevity, a detailed formalisation for prefixing the subinstance identifier to each atomic element of a substituted remote condition expression (i.e. the adaptation to the namespace by the function $ns^{\mathfrak{CE}} : \mathfrak{ID} \times \mathfrak{CE} \to \mathfrak{CE}$) is omitted. Furthermore, the above case differentiations are abbreviated such that, e.g. (l, op, r) denotes a tuple pattern representing the case $c \in \mathfrak{CE} \times \mathcal{OP} \times \mathfrak{CE}$ or (r, l) in case $c \in \mathfrak{Ref}$.

Finally, the substitution function for Probability statements is defined

$$
{}^{c}\!/\!\phi^{Probability} : P, C, I \mapsto \bigcup_{pr \in Pr} \begin{cases} (pr.l, {}^{c}\!/\!\phi(pr.c, C, I)) & \text{if } pr \in \mathfrak{Prob}_S \\ (pr.l, {}^{c}\!/\!\phi(pr.c, C, I), pr.t) & \text{if } pr \in \mathfrak{Prob}_T \end{cases}
$$

3.3.3 Guard Expansion

This section provides the transformation semantics for resolving reactive expressions used in forward and guards statements. It describes how the combinatorics arising from multiple *conditional reactive*s and addressed conditional forward statements is handled.

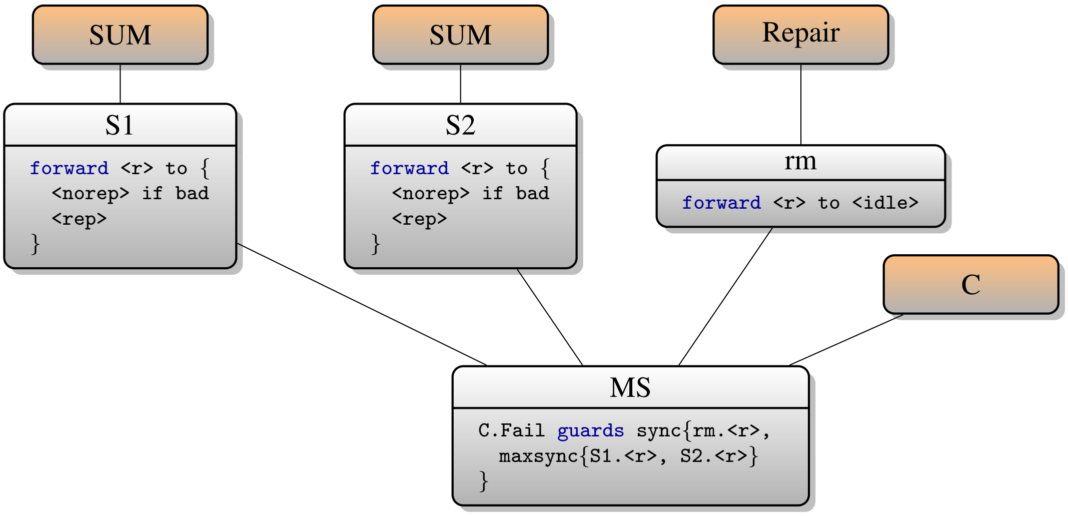
In order to illustrate the following transformation definitions, the MS example is revisited and adapted. Figure 3.10 represents the MS model before applying this transformation. As it can be seen, forward statements are addressed at intermediate levels. In a manner likewise to the preceding section, indirections over intermediate levels will be eliminated to end up with resolved guards statements, solely composed of elements which directly refer to statements of Behavior instances. This third transformation step ultimately outputs a LARES$_{\text{BASE}}$ model.

For this purpose, the traversal function τ_{ge} is defined to perform the *guard expansion* transformation. It resolves reactive expressions and thereby expands forward statements in order to finally construct resolved expanded guards statements:

$$\tau_{ge} : \mathfrak{I}_{\mathcal{M}} \times \mathfrak{FW}_{ge} \rightarrow \mathfrak{BW}_{ge}$$

As in the preceding sections, a forward function ζ_{ge} and a backward function β_{ge} have to be defined. Alike the condition expansion transformation, no information is required to be forwarded such that the empty set $\emptyset$ is the only propagated value:

$$\mathfrak{FW}_{ge} := \{\emptyset\}$$



```
forward <r> to {
  <norep> if bad
  <rep>
}
```

```
forward <r> to {
  <norep> if bad
  <rep>
}
```

```
forward <r> to <idle>
```

```
C.Fail guards sync{rm.<r>,
  maxsync{S1.<r>, S2.<r>}
}
```

Figure 3.10: Modified MS example: Expansion of reactive expressions

Since expanded forward and guards statements are contained by reconstructed subinstances, the backward information encompasses no further information apart from the subinstances:

$$\mathfrak{BW}_{ge} := \mathcal{I}_{\mathcal{M}}$$

Forward Propagation

According to the above mentioned fact that the expansion of forward and guards statements only depends on the results of the resolved forward statements of the subinstances, no information is required to be propagated in the direction towards the leaves. The forward function ζ_{ge} is thus, as it was in the preceding section, defined as the identity function:

$$\zeta_{ge} : (i, fw) \mapsto (i, \emptyset)$$

Backward Aggregation

A given instance i is resolved by using the backward information $BW \in \mathfrak{BW}_{ge}$ obtained by the preceding recursive step of the guards expansion transformation performed on the subinstances. Accordingly, the corresponding backward function is given by

$$\beta_{ge} : (i, fw, BW) \mapsto (i')$$

The instance i' which contains only resolved guards statements can be constructed by using the function $b_{ge} : \mathfrak{M}_{body} \times \mathcal{P}(\mathcal{I}_{\mathcal{M}}) \rightarrow \mathfrak{M}_{body}$ on i's body. Let $l = i.t.l$ be the name of i's Module definition, $d = i.t.d$ be the set of delegates and $b = i.t.b$ be the body. The subinstance i' is recomposed (with the modified body whose local and remote references to forward statements were resolved) in accordance with

$$i' := (i.l, \underbrace{(l, \emptyset, d, b_{ge}(b, BW))}_{\in \mathcal{M}}, ())$$

Similar to Condition statements, forward statements may also have local interdependencies which will only be resolved if the processing follows a topological order. Hence again, a function $ts : \mathcal{P}(\mathcal{F}) \rightarrow Seq(\mathcal{F})$ is declared for the purpose of determining an order in accordance with local interdependencies among forward statements. A stepwise substitution function ${}^{\mathrm{f}}\!/\!\!\phi^{forward}$ can then be used which requires three arguments: A sequence of unresolved local forwards, a set of already resolved local forwards and a set of processed subinstances:

$$\mathfrak{f}\!/\!\phi^{forward} : \underbrace{Seq(\mathcal{F})}_{\substack{\text{unresolved}\\ \text{forwards}}} \times \underbrace{\mathcal{P}(\mathcal{F})}_{\substack{\text{resolved}\\ \text{local}\\ \text{forwards}}} \times \underbrace{\mathcal{P}(\mathcal{I})}_{\substack{\text{processed}\\ \text{subinstances}}} \rightarrow \underbrace{\mathcal{P}(\mathcal{F})}_{\substack{\text{resolved}\\ \text{forwards}}}$$

Eventually, the resolved forward statements F_{res} can be determined by initially applying ts to obtain a sound sequence:

$$F_{res} = \mathfrak{f}\!/\!\phi^{forward}(ts(b.F), \emptyset, BW)$$

The resolved forward statements are subsequently used to compose a body for substitution. Therefore, let

$$\mathfrak{f}\!/\!\phi^{guards} : \underbrace{\mathcal{P}(\mathcal{G})}_{\substack{\text{unresolved}\\ \text{guards}}} \times \underbrace{\mathcal{P}(\mathcal{F})}_{\substack{\text{resolved}\\ \text{local}\\ \text{forwards}}} \times \underbrace{\mathcal{P}(\mathcal{I}_{\mathcal{M}})}_{\substack{\text{processed}\\ \text{subinstances}}} \rightarrow \underbrace{\mathcal{P}(\mathcal{G})}_{\substack{\text{resolved}\\ \text{guards}}}$$

be a function declaration that resolves guards statements by using given local and remote (resolved) forward statements. The body transformation function b_{ge} is then defined as

$$b_{ge} : (b, I) \mapsto \underbrace{(\emptyset, \emptyset, I, \emptyset, \mathfrak{f}\!/\!\phi^{guards}(b.G, F_{res}, BW), F_{res}, b.ic, \emptyset, b.P)}_{\text{the modified body (where the referred forward statements have been substituted)}}$$

It remains to define the previously declared functions $\mathfrak{f}\!/\!\phi^{forward}$ and $\mathfrak{f}\!/\!\phi^{guards}$. Firstly, the substitution function $\mathfrak{f}\!/\!\phi^{forward}$ of the forward statements is recursively defined such that forward statements are taken from a given sequence F and resolved as long as $F \neq ()$. The resolved forward statements are collected and used for subsequent resolution steps:

$$\mathfrak{f}\!/\!\phi^{forward} : (F, F_{res}, I) \mapsto \begin{cases} F_{res} & \text{if } F = () \\ \mathfrak{f}\!/\!\phi^{forward}(F_{tail}, \{f'_{head}\} \cup F_{res}, I) & \text{else,} \end{cases}$$

where

- $F = (f_{head}) \circ F_{tail}$ splits the sequence F into a head element f_{head} and the remaining sequence F_{tail}. The topological order of F ensures that the forward statement f_{head} is independent from the remaining elements contained in F_{tail}.
- $f'_{head} = (f_{head}.c, f_{head}.l, CR)$ constructs a resolved forward which is composed of the original condition, the original label and a multiset CR to capture possibly identical resolved *conditional reactive*s. CR is processed by the function

$$\mathfrak{f}\!/\!\phi^{cr} : \mathfrak{CR} \times \mathcal{P}(\mathcal{F}) \times \mathcal{I}_{\mathcal{M}} \rightarrow \mathcal{P}(\mathfrak{CR})$$

which performs the substitution of a single *conditional reactive* by taking the set of resolved local forwards statements F_{res} and those contained in the processed subinstances into account:

$$CR = \bigcup [\, {}^{\mathbf{f}}\!/\!\phi^{cr}(cr, F_{res}, I)^k \mid cr \in^k f_{head}.CR \,]$$

As described above, the substitution function ${}^{\mathbf{f}}\!/\!\phi^{cr}$ is responsible for the substitution of a single *conditional reactive* element cr. Based on multiple choices among the referred local and remote guards/forward labels, combination of substitutions lead to a number of *conditional reactive*, contained by CR'. The function is hence defined by

$${}^{\mathbf{f}}\!/\!\phi^{cr} : (cr, R_{res}, I) \mapsto CR'$$

Let RF include all referable (local and remote resolved) forward statements and their references:

$$RF := \underbrace{\left\{ \overbrace{(((), f.l), f)}^{\in \mathfrak{Ref} \times \mathfrak{F}} \mid f \in F_{res} \right\}}_{\text{resolved local forwards}} \cup \underbrace{\left\{ \overbrace{(((i.l), f.l), f)}^{\in \mathfrak{Ref} \times \mathfrak{F}} \mid i \in I \wedge f \in i.F \right\}}_{\text{resolved remote forwards}}$$

In addition to that, let a conditional reactive cr be denoted as a tuple which consists of a condition expression ce and a reactive expression re, i.e. $cr = (ce, re)$, and let the function $dep : \mathfrak{Ref} \times \mathfrak{RE} \rightarrow \mathbb{B}$ be given which determines whether a reference r is used by the given *reactive* expression. The set of actually referred forward statements can be determined by

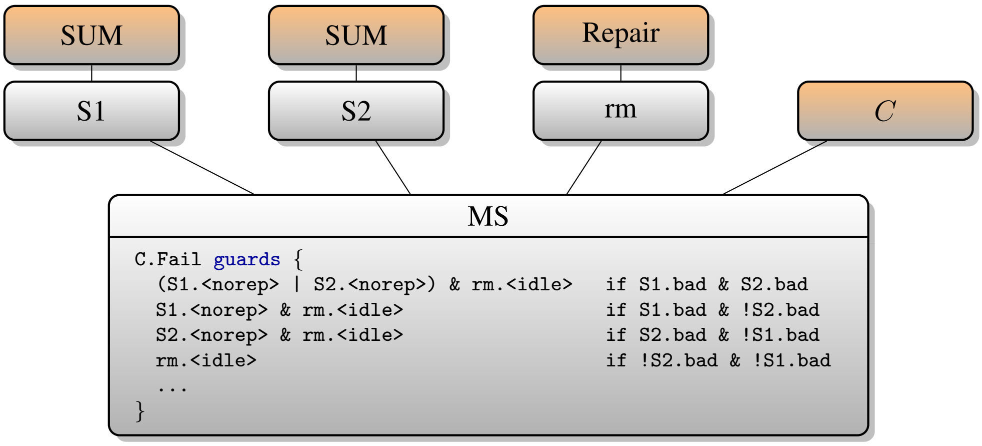
$$RF' = \{(r, f) \in RF \mid dep(r, re)\}$$

The global condition $f.c$ and the restrictive conditions rc of an addressed forward statement f inside RF' are then merged and simplified in order to obtain a set of multisets of referred conditional reactives RCR addressed via a reference r:

$$RCR = \left\{ \left[(r, simplify(f.c \wedge rc), re)^k \;\middle|\; (rc, re) \in^k f.CR \right] \quad \mid (r, f) \in RF' \right\}$$

As a consequence of the fact that conditional reactives inside a single forward statement are competing (i.e. a choice can be made among them), all possible combination of choices need to be constructed by the cross-product over all of these sets:

$$RCRC = \prod_{rcr \in RCR} rcr$$



```
C.Fail guards {
  (S1.<norep> | S2.<norep>) & rm.<idle>    if S1.bad & S2.bad
  S1.<norep> & rm.<idle>                   if S1.bad & !S2.bad
  S2.<norep> & rm.<idle>                   if S2.bad & !S1.bad
  rm.<idle>                                if !S2.bad & !S1.bad
  ...
}
```

Figure 3.11: Modified MS example: Expanded guard statement

Let $rcrc \in RCRC$ be a combination of referred conditional reactives. The set of reactive expressions of unconditional reactives (i.e. arising from those conditional expressions that do not comprise a condition expression) is given by

$$R_U = \{(r, ce, re) \in rcrc \mid ce = true\}$$

whereas the set of genuine *conditional reactive*s is given by the complement

$$R_C = \{(r, ce, re) \in rcrc \mid ce \neq true\}$$

For the example given in Figure 3.10, the combinations arising from choices of referred *conditional reactive*s are shown as follows:

$$\begin{bmatrix} \texttt{S1}.\langle\texttt{r}\rangle, \texttt{bad}, \langle\texttt{norep}\rangle \\ \texttt{S1}.\langle\texttt{r}\rangle, \texttt{true}, \langle\texttt{rep}\rangle \end{bmatrix} \times \begin{bmatrix} \texttt{S2}.\langle\texttt{r}\rangle, \texttt{bad}, \langle\texttt{norep}\rangle \\ \texttt{S2}.\langle\texttt{r}\rangle, \texttt{true}, \langle\texttt{rep}\rangle \end{bmatrix} \times \begin{bmatrix} \texttt{rm}.\langle\texttt{r}\rangle, \texttt{true}, \langle\texttt{idle}\rangle \end{bmatrix}$$

For illustratory reasons, let $rcrc$ be one of these combinations used to derive R_U and R_C:

$$\left\{ \begin{array}{l} (\texttt{S1}.\langle\texttt{r}\rangle, \texttt{bad}, \langle\texttt{norep}\rangle), \\ (\texttt{S2}.\langle\texttt{r}\rangle, \texttt{bad}, \langle\texttt{norep}\rangle), \\ (\texttt{rm}.\langle\texttt{r}\rangle, \texttt{true}, \langle\texttt{idle}\rangle) \end{array} \right\} \Rightarrow \begin{array}{l} R_U = \{(\texttt{rm}.\langle\texttt{r}\rangle, \texttt{true}, \langle\texttt{idle}\rangle)\} \\ R_C = \{(\texttt{S1}.\langle\texttt{r}\rangle, \texttt{bad}, \langle\texttt{norep}\rangle), (\texttt{S2}.\langle\texttt{r}\rangle, \texttt{bad}, \langle\texttt{norep}\rangle)\} \end{array}$$

This single combination will finally result in the outcome, shown in Figure 3.11, which covers the distinct cases of the genuine conditional reactives. Following the described scheme, the set of substituted and expanded *conditional reactive*s CR' can hence be derived by considering all combinations of choices and all subsets of *conditional reactive* expressions (coming from the genuine conditional reactives).

Apart from these combinations, all guarded transitions inside all Behavior instances have to be taken into account (as atomic reactions). Hereby, the first element of a contained tuple represents the reference (i.e. the addressed Behavior instance and the guard label). The second element is $true$, primarily because this label is always offered and secondly because it has to conform with R_U and R_C. The third element represents the atomic resolved guard label reference which includes the addressed distribution type:

$$R_B = \{\ (\underbrace{(d.l,t.l)}_{\in\mathfrak{Ref}}, true, \underbrace{(d.l,t.l,\pi_1(t.d))}_{\substack{\text{atomic resolved}\\ \text{guard label reference}}}) \mid \underbrace{d \in i.t.d}_{\substack{\text{Behavior}\\ \text{instances}}} \wedge \underbrace{t \in d.t.b.T : t.l \neq ()}_{\substack{\text{guarded}\\ \text{transitions}}}\ \}$$

The prerequisite for a valid model is that for each guard label offered by a Behavior definition, the providing guarded transitions are required to have the same distribution type. This ensures that R_B has only one entry for a single reference $(b.l, t.l)$.

Each conditional reactive is composed of a condition ce' and a reactive expression re. The latter is substituted by a function $\mathfrak{f}/\phi^{\mathfrak{RE}}$ (which will be defined shortly after):

$$CR' = \underbrace{\bigcup_{rcrc \in RCRC} \underbrace{\left(\bigcup_{R'_C \in \mathcal{P}(R_C)} (ce', \mathfrak{f}/\phi^{\mathfrak{RE}}(re, R'_C \cup R_U \cup R_B)) \right)}_{\text{the distinct cases arising from the genuine conditional reactives}}}_{\text{the combinations arising from choices}}$$

The function $ns^{\mathfrak{RE}} : \mathfrak{ID} \times \mathfrak{RE} \to \mathfrak{RE}$ is subsequently used for prefixing the given identifier to all atomic elements of the expression. Its detailed definition is omitted for reasons of brevity. This allows defining the function $\mathfrak{f}/\phi^{\mathfrak{RE}}$ which resolves the references of a reactive expression r by their addressed local and remote forward statements R:

$$\mathfrak{f}/\phi^{\mathfrak{RE}} : (r, R) \mapsto \begin{cases} \mathfrak{f}/\phi^{\mathfrak{RE}}(o_1, R) \wedge \ldots \wedge \mathfrak{f}/\phi^{\mathfrak{RE}}(o_n, R) & \text{if } r = sync\{o_1, \ldots o_n\} \\ \mathfrak{f}/\phi^{\mathfrak{RE}}(o_1, R) \vee \ldots \vee \mathfrak{f}/\phi^{\mathfrak{RE}}(o_n, R) & \text{if } r = maxsync\{o_1, \ldots o_n\} \\ \mathfrak{f}/\phi^{\mathfrak{RE}}(o_1, R) \wedge \neg \ldots \wedge \neg \mathfrak{f}/\phi^{\mathfrak{RE}}(o_n, R) \vee \\ \neg \mathfrak{f}/\phi^{\mathfrak{RE}}(o_1, R) \wedge \mathfrak{f}/\phi^{\mathfrak{RE}}(o_2, R) \wedge \neg \ldots \vee \\ \ldots \vee \\ \neg \ldots \wedge \neg \mathfrak{f}/\phi^{\mathfrak{RE}}(o_{n-1}, R) \wedge \mathfrak{f}/\phi^{\mathfrak{RE}}(o_n, R) & \text{if } r = choose\{o_1, \ldots o_n\} \\ ns^{\mathfrak{RE}}(r.i.l, re) & \text{if } (r, ce, re) \in R \\ r & \text{else.} \end{cases}$$

In order to construct the condition ce', let $C_{all} := \{\pi_2(r_c) \mid r_c \in R_C\}$ be the set of all conditions inside a choice-combination $rcrc$ and let $C_{sub} := \{\pi_2(r_c) | r_c \in R'_C\}$ be the subset of conditions determined by the elements of R'_C of the power-set over R_C. By doing so, the new conditional expression is finally constructed by the given choice-combination:

$$ce' = ce \wedge \left(\bigwedge_{x \in C_{sub}} x \right) \wedge \left(\bigwedge_{x \in C_{all} \setminus C_{sub}} \neg x \right)$$

While performing the expansion over the tree structure of the model, starting at the leaf instances, all associated `guards` and `forward` statements will be resolved. When reaching the root node of the instance tree, all `guards` statements are considered *resolved*. This means that they directly refer to guard labels of instantiated behaviours.

For the running example, the outcome is straightforward due to the absence of conditional forwards (i.e. the Boolean expression is implicitly $true$). The two guards statements of the network instance are resolved by the forwards associated with the link instances, resulting in

```
(l[1].BLink.lfailed & l[2].BLink.lok)
  guards l[2].BLink.⟨swactive⟩
((l[1].BLink.lfailed & l[2].BLink.lfailed) & l[3].BLink.lok)
  guards l[3].BLink.⟨swactive⟩
```

On the contrary, the following guards statement, which is stated inside the System definition, remains untouched as `BSys.<systemfail>` is already resolved. Consequently, the whole guards statement is considered *resolved*:

```
((p[1].BProcessor.pfailed|p[2].BProcessor.pfailed) |
  ( n.l[1].BLink.lfailed &
    n.l[2].BLink.lfailed &
    n.l[3].BLink.lfailed )) guards BSys.⟨systemfail⟩
```

Lastly, it remains to define the function $\text{f}\!/\!\phi^{guards}$. It applies the function $\text{f}\!/\!\phi^{cr}$ in order to substitute the references by the addressed resolved conditional reactives:

$$\text{f}\!/\!\phi^{guards} : (G, F_{res}, I) \mapsto \bigcup [\underbrace{(\{g.g\} \times \text{f}\!/\!\phi^{cr}(cr, F_{res}, I) \times \{()\})}_{\text{resolved guards statements}}{}^{m \cdot n} | \, g \in^m G \wedge cr \in^n g.CR]$$

As a final remark, neither the Condition statements nor the forward statements are of any further avail and can be removed from the model, as their information is now represented inside the resolved guards statements. The process of their removal (with the purpose of obtaining a LARES$_{\text{BASE}}$ model in the end) is not mentioned in this work, as it is considered trivial.

3.4 From LARES$_{\text{BASE}}$ to LTS as Target Formalism

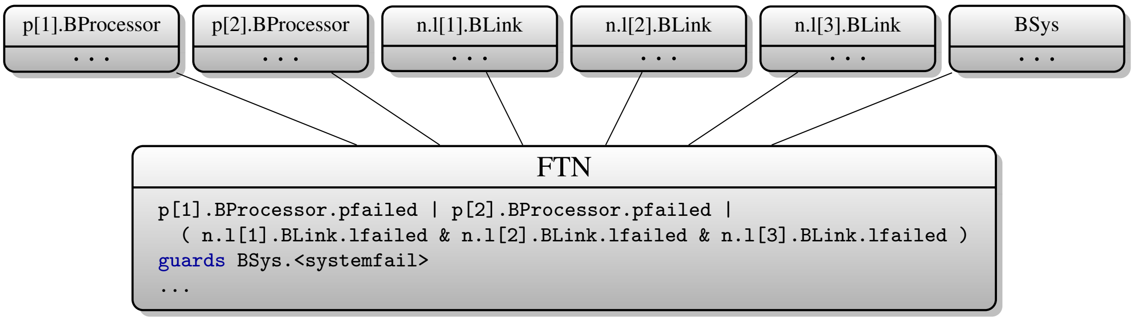
This section specifies how a LARES$_{\text{BASE}}$ model is transformed into a transition system. Within the following three subsections, a hierarchy resolved variant of LARES$_{\text{BASE}}$ called LARES$_{\text{FLAT}}$ will be defined (cf. Section 3.4.1), furthermore the formal representation of an arbitrary behaviour (like a Markov Chain) will be formalised (cf. Section 3.4.2) and subsequently be used as the target formalism of a transformation from LARES$_{\text{FLAT}}$ (cf. Section 3.4.3).

3.4.1 LARES$_{\text{FLAT}}$: A Hierarchy-Resolved LARES$_{\text{BASE}}$

A LARES$_{\text{FLAT}}$ model is obtained from a LARES$_{\text{BASE}}$ model by resolving its hierarchy. As a result, a LARES$_{\text{FLAT}}$ model solely contains the system instance which comprises all aggregated Behavior instances and guards statements.

The formal definition restricts the general LARES representation to the following form:

$$LARES_{FLAT} := \{(B, M, \underbrace{(l, \overbrace{(l, de, b, ())}^{\in \mathcal{M}}, ic)}_{\in \mathcal{I}}) \in LARES \mid B = \emptyset \wedge M = \emptyset \wedge ic = ()\}$$



```
p[1].BProcessor.pfailed | p[2].BProcessor.pfailed |
  ( n.l[1].BLink.lfailed & n.l[2].BLink.lfailed & n.l[3].BLink.lfailed )
guards BSys.<systemfail>
...
```

Figure 3.12: LARES$_{\text{FLAT}}$ model for the FTN example

The root instance will only contain the set of delegations de (which in turn consists of Behavior instances) and the set of all resolved guards statements. No other statements or definitions will remain. The LARES$_{\text{FLAT}}$ representation of the FTN example is partially depicted by Figure 3.12. The associated transformation from LARES$_{\text{BASE}}$ to LARES$_{\text{FLAT}}$ is straight forward: Successively, the instance names have to be prefixed inside for each Behavior instance and inside each element of a guards statement. Subsequently, the Behavior instances and the guards statements are added to the parent of the current instance, while the current instance can then be deleted. By doing so, the hierarchy is successively resolved, starting at the leaves, continuing until the root instance is reached and ending up in the above LARES$_{\text{FLAT}}$ representation. Finally, the labels of the elements implicitly capture the structure of the model, whereas the explicit hierarchical structure is eliminated. Due to its simplicity and transferability, only specific parts of the transformation are formalised in this work. The traversal function τ_{hr} performs the hierarchy resolution transformation of Module instance by successively concatenating the current instance identifier in order to construct a namespace for each element. This derives a LARES$_{\text{FLAT}}$ representation which only consists of a (hierarchy resolved) System instance:

$$\tau_{hr} : \mathcal{I}_{\mathcal{M}} \times \mathfrak{FW}_{hr} \rightarrow \mathfrak{BW}_{hr}$$

As no forward information is required to resolve the hierarchy, the forward information only encompasses the empty set.

$$\mathfrak{FW}_{hr} := \{\emptyset\}$$

Equivalently to the preceding transformations, a Module instance is relayed backwards. Therefore, the backward information is defined by

$$\mathfrak{BW}_{hr} := \mathcal{I}_{\mathcal{M}}$$

Forward Propagation

As the hierarchy resolution process only depends on the results of hierarchy resolved substructures, the forward function ζ_{hr} is defined as the identity function:

$$\zeta_{hr} : (i, fw) \mapsto (i, \emptyset)$$

Backward Aggregation

The backward function β_{hr} resolves a given instance i using the (hierarchy resolved) instances $BW \in \mathfrak{BW}_{hr}$ which have been relayed backwards by the preceding recursive step of the transformation. It is defined by

$$\beta_{hr} : (i, fw, BW) \mapsto (i').$$

Subsequently, two function will be used in order retrieve the Behavior instances DE_{sub} and the guards statements G_{sub} from the subinstances. Hereby, the function $\mathbf{h}/\!\!\!\phi^{\mathfrak{DE}} : \mathcal{I}_{\mathcal{M}} \rightarrow \mathcal{P}(\mathcal{I}_{\mathcal{B}})$ modifies the delegates of a given subinstance such that the name of the subinstance is prefixed to the identifiers of its delegates. This function is applied to all child instances in order to reduce one hierarchy level in the subtree. The function ${}^{\mathbf{h}}\!/\!\phi^{\mathcal{G}} : \mathcal{I} \to mset(\mathcal{G})$ by contrast modifies all guards statements of the subinstances and aggregates them.

$$DE_{sub} = \bigcup_{i_{sub} \in BW} {}^{\mathbf{h}}\!/\!\phi^{\mathfrak{DE}}(i_{sub}) \qquad G_{sub} = \bigcup_{i_{sub} \in I_{sub}} {}^{\mathbf{h}}\!/\!\phi^{\mathcal{G}}(i_{sub})$$

The resulting hierarchy resolved Module instance i' is composed as follows:

$$i' := (i.l, \underbrace{(i.t.l, \emptyset, i.t.d \cup DE_{sub}, b')}_{i\text{'s modified Module definition}}, i.ic),$$

$$\text{where } b' = \underbrace{(\emptyset, \emptyset, \emptyset, \emptyset, i.t.b.G \cup G_{sub}, \emptyset, \emptyset, \emptyset, i.t.b.P)}_{i\text{'s modified Module body}}$$

The hierarchy resolution function ${}^{\mathbf{h}}\!/\!\phi^{\mathfrak{DE}}$ (which collects and modifies all Behavior delegates) and the hierarchy resolution function ${}^{\mathbf{h}}\!/\!\phi^{\mathcal{G}}$ (which collects and modifies all guards statements) are defined by

$${}^{\mathbf{h}}\!/\!\phi^{\mathfrak{DE}} : i \mapsto \{\, {}^{\mathbf{h}}\!/\!\phi^{d}(i.l, d) \mid d \in i.t.d\} \quad \text{and} \quad {}^{\mathbf{h}}\!/\!\phi^{\mathcal{G}} : i \mapsto [\, {}^{\mathbf{h}}\!/\!\phi^{g}(i.l, g)^{k} \mid g \in^{k} i.t.b.G\}$$

The function for building the namespace of a single Behavior delegate is defined by

$${}^{\mathbf{h}}\!/\!\phi^{d} : (l, d) \mapsto (l \circ d.l, d.t, d.ic)$$

Establishing the namespace for a guards statement turns out to be more complicated as each contained expression has to be modified:

$${}^{\mathbf{h}}\!/\!\phi^{g} : (l, g) \mapsto \big(ns^{\mathfrak{CE}}(l, g.c), \big\{\, (ns^{\mathfrak{CE}}(l, c), ns^{\mathfrak{RE}}(l, r) \mid (c, r) \in g.CR\big\}\,, (l) \circ g.ns)\big)$$

3.4.2 Abstract Representation of a Transition Systems

A transition system represents some discrete behaviour in terms of states and transitions. Let the universal set of transitions be defined by $\mathcal{T} := \mathcal{S} \times \mathcal{E} \times \mathcal{S}$. Hereby, $\mathcal{S}$ denotes the universal set of enumerable states and $\mathcal{E}$ represents the universal set of extensions (which may e.g. include a label and a distribution). A transition is thus composed of a source state, some kind of extension attribute and a target state. An *extended* transition system is given by the tuple $(T, s_0) \in \mathcal{TRA}$ which contains a multiset of transitions $T \subset mset(\mathcal{T})$ and the initial state $s_0 \in \mathcal{S}$. T is defined as a multiset in order to capture choices between transitions that lead to the same target state (apart from providing the same transition annotations, such as a label or a given distribution).

The transition system is called an *extended stochastic labelled transition system* (ESLTS), when the type of transition definition incorporates a label l and a delay that is either zero (immediate) with weight w or exponential (Markovian) with rate r (as given by $\mathcal{E} := \Sigma^{+} \times (\mathcal{D}_{exp} \cup \mathcal{D}_{imm})$), where the sets of possible distributions are defined by

$$\mathcal{D}_{exp} := \{\xrightarrow{r} | r \in \mathbb{R}\} \text{ and } \mathcal{D}_{imm} := \{\overset{w}{\dashrightarrow} | w \in \mathbb{R}\}$$

Accordingly, $\mathcal{T}_{\mathcal{M}} := \{(s_s, (l, \xrightarrow{r}), s_t) \in \mathcal{T}\}$ denotes the set of labelled Markovian transitions and $\mathcal{T}_{\mathcal{I}} := \{(s_s, (l, \overset{w}{\dashrightarrow}), s_t) \in \mathcal{T}\}$ denotes the set of labelled weighted immediate transitions, where $\mathcal{T}_{\mathcal{M}} \dot{\cup} \mathcal{T}_{\mathcal{I}} = \mathcal{T}$.

In order to relate a source state and its outgoing transitions, the function $\delta : \mathcal{S} \rightarrow mset(\mathcal{T})$ is declared (its definition will be omitted here).

3.4.3 LTS Semantics for LARES$_{\text{FLAT}}$

A sound formal execution semantics is essential in order to perform simulation or reachability analysis of a LARES model. In this section, the execution semantics for LARES is given by defining a transformation into a transition system.

Let $f \in LARES_{FLAT}$ be a LARES$_{\text{FLAT}}$ model which contains the system instance i_1, obtained by $i_1 = f.i_1$. The set $B_f = i_1.t.d$ then comprises all Behavior instances and $G_f = i_1.t.b.G$ contains all resolved guards statements. Let $n = |B_f|$ be the number of Behavior instances. The Behavior instances can be written as a tuple

$$(b_1, b_2, \dots, b_n), \text{ where } b_i \in B_f.$$

In order to clarify the above: Each instantiated Behavior has a specific namespace originating from the hierarchy resolution process. Here and subsequently, the index of each entry in the above tuple of Behavior instance encodes the namespace of a Behavior instance.

Each Behavior instance contains a set of states S_i which can be obtained for all instances via the projection $.t.S$ on each element of $(b_1, \dots, b_n)$:

$$(S_1, S_2, \dots, S_n) = (b_1.t.S, b_2.t.S, \dots, b_n.t.S)$$

The *potential state space* S arising thereof is given by the cross product of the sets of states of all Behavior instances:

$$S = S_1 \times S_2 \times \ldots \times S_n.$$

A *composed state* $s \in S$ is subsequently denoted as a tuple of states:

$$(s_1, s_2, \dots, s_n) = s, \text{ where } s_i \in S_i$$

For each system, an initial composed state $s^1 \in S$ needs to be given that is determined by the initial configuration of all instantiated Behaviors:

$$s^1 = (b_1.t.ic.l, b_2.t.ic.l, \dots, b_n.t.ic.l)$$

The set of states of a Behavior instance b_i can be obtained by applying the projection function $.t.b.T$. This is done for each element of the tuple of Behavior instances in order to obtain a tuple of sets of transitions:

$$(T_1, T_2, \dots, T_n) = (b_1.t.b.T, b_2.t.b.T, \dots, b_n.t.b.T)$$

Let the function $succ$ be declared which calculates the successor s' of a composed state tuple s. Let a set of jointly acting transitions T, where each transition is performed by a distinct Behavior instance, be given and let a function $\xi : \mathcal{P}(\mathcal{T}) \times \mathcal{S} \rightarrow \mathcal{S}$ be declared. This function determines which of the component states of s have to be switched to their successor states via matching transitions:

$$succ : (T, s) \mapsto (\xi(T, s_1), \dots, \xi(T, s_n)), \text{ where } s = (s_1, \dots, s_n).$$

In accordance with the above, the function ξ calculates the successor of a component source state s_i and a given set of transitions T. Hereby, either the target state $t.t$ in case of a matching transition $t \in T$ or otherwise the source state s_i will become the succeeding component state:

$$\xi : (T, s_i) \mapsto \begin{cases} t.t & \text{if } \exists t \in T : t.s \overset{\wedge}{=} s_i \\ s_i & \text{else.} \end{cases}$$

In order to subsequently determine the composed successor state, an abbreviation notation is used which is defined by $\langle T, s \rangle = succ(T, s)$.

The Execution Semantics of Unguarded Behaviour

The execution semantics is described by Structural Operational Semantics (SOS) rules which were introduced in [113]. As stated in [1], this representation has an intuitive appeal and is widely used as a flexible framework to give operational semantics to programming

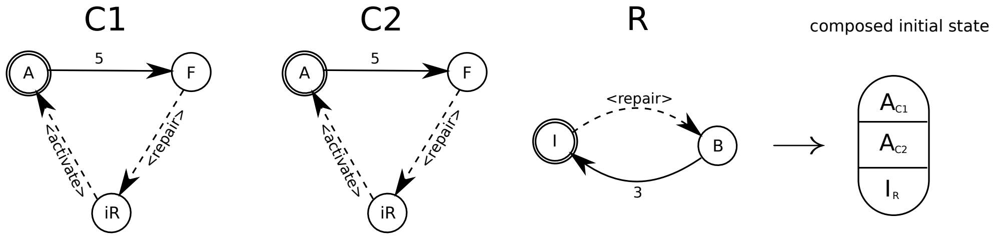


Figure 3.13: Example: Composed initial state

and specification languages. Each rule defines a number of antecedent conditions and some side-conditions that imply a certain consequence:

$$\frac{\langle antecedent\rangle *}{\langle consequence\rangle} \quad \langle side\ condition\rangle$$

If a Behavior instance b_i can perform an unguarded Markovian transition $s_i \xrightarrow{\lambda} s_i' \in T_i$ from state $s_i \in S_i$ into $s_i' \in S_i$, the component state s_i of the composed state s will also change to s_i'. The SOS rule for an unguarded Markovian transition is hence given by

$$\frac{s_i \xrightarrow{\lambda} s_i'}{s \xrightarrow{\lambda} s'} \qquad s' = \langle\{s_i \xrightarrow{\lambda} s_i'\}, s\rangle \tag{3.2}$$

A similar rule captures the case of being able to performing an unguarded immediate transition $s_i \overset{w}{\dashrightarrow} s_i' \in T_i$:

$$\frac{s_i \overset{w}{\dashrightarrow} s_i'}{s \overset{w}{\dashrightarrow} s'} \qquad s' = \langle\{s_i \overset{w}{\dashrightarrow} s_i'\}, s\rangle \tag{3.3}$$

In Figure 3.13, an example system composed of the Behavior instances `C1`, `C2` and `R` is shown including the derived composed initial state. As depicted in Figure 3.14, the first SOS rule (3.2) was applied to the composed initial state in order to determine the succeeding composed states. Due to the absence of unguarded immediate transitions in the given example, rule (3.3) was not applied.

The Execution Semantics of Guarded Behaviour

The execution of guarded transitions depends on guards statements which specify interaction among the instances. To illustrate the complex interaction mechanism, the example introduced in Figure 3.13 is consulted again. The model is yet incomplete in the sense that there are no guards statements given. As it was mentioned beforehand, these statements

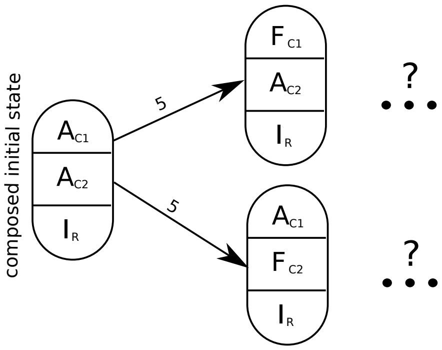


Figure 3.14: Example: Semantics for unguarded transitions

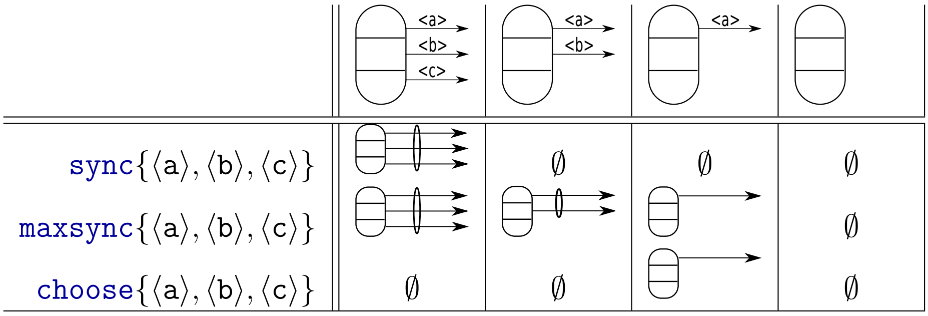


Figure 3.15: Exemplified illustration of the reactive operator semantics

specify how instantiated Behaviors interact. Let the following guards statement be added to the example model:

$$\begin{array}{ll}(g1) & F_{C1} \wedge I_R \; \mathit{guards} \; \{ \\ & \quad \text{if } \mathit{true} \quad \langle \mathit{repair} \rangle_{C1} \vee \langle \mathit{repair} \rangle_R \\ & \} \end{array}$$

The generative part requires component $C1$ to be in the failed state F and the repairman R is required to be in the idle state I. If the generative part is fulfilled, an event will be generated that may influence $C1$ and R depending on whether the reactive expression is satisfied (which means that the addressed behaviours reveal the required behaviour). According to the given example, either $C1$ or R should be able to perform a $\langle \mathit{repair} \rangle$ behaviour or both at once. Since the $\vee$-operator originates from a $\mathit{maxsync}$ within the reactive expression, the semantics can be established from studying the synopsis given in Figure 3.15 which depicts the synchronisation semantics for the specific operators of a reactive expression.

As shown in Figure 3.14, the composed state (F_{C1}, A_{C2}, I_R) which satisfies the generative expression of the guards statement $(g1)$ is reached:

$$(F_{C1}, A_{C2}, I_R) \vDash F_{C1} \wedge I_R$$

The reactive part has to be considered next. In the preceding transformation, the reactive expression was translated to its logical expression equivalent $\langle repair\rangle_{C1} \vee \langle repair\rangle_R$, by replacing the maxsync operator by a logical disjunction of its arguments, which is achieved by using the infix operator $\vee$. The incurred logical expression is transformed into a product term which, to be more precise, is a minterm representation (i.e. a product term in which each variable appears once) in order to comply with the state space (which is a result of the SPA transformation approach described in Section 3.5) in a later step:

$$(\langle repair\rangle_{C1} \wedge \langle repair\rangle_R) \;\vee\; (\overline{\langle repair\rangle_{C1}} \wedge \langle repair\rangle_R) \;\vee\; (\langle repair\rangle_{C1} \wedge \overline{\langle repair\rangle_R})$$

A behaviour represented by a guarded transition which is labelled $\langle repair\rangle$, will be revealed by both instantiated behaviours $C1$ if the failed state F is current and R if the idle state I is current. Accordingly, one of the above product terms is satisfied by the considered composed state:

$$(F_{C1}, A_{C2}, I_R) \vDash \langle repair\rangle_{C1} \wedge \langle repair\rangle_R$$

For all guard label references inside the maxsync operator, a corresponding behaviour can be found in state (F_{C1}, A_{C2}, I_R). This matches the second row and the first column of Figure 3.15. For this reason, the addressed transitions will perform synchronously. The example model is completed by including two further guards statements:

$$(g2)\quad F_{C2} \wedge I_R \; guards \; \{ \quad \text{if } true \;\; \langle repair\rangle_{C2} \vee \langle repair\rangle_R \quad \}$$

$$(g3)\quad I_R \; guards \; \{ \quad \text{if } true \;\; \langle activate\rangle_{C1} \vee \langle activate\rangle_{C2} \quad \}$$

The successive application of the initially given rules and the scheme on how to deal with guards statements until a fixed point is reached will exhaustively construct the state space (cf. Figure 3.16).

There may also be choices between different behaviours within a component and between multiple conditional reactives of a single guards statement, which is not covered by the above illustrations. These aspects are captured in the next section in which the general execution semantics for the interaction behaviour will be defined.

General Execution Semantics of Interaction Behaviour

Each guards statement in the multiset of guards statements G_f may be composed of one or more conditional reactives. For each conditional reactive, a multiset of tuples is constructed within which each tuple contains a condition expression (built from the conjunction of the

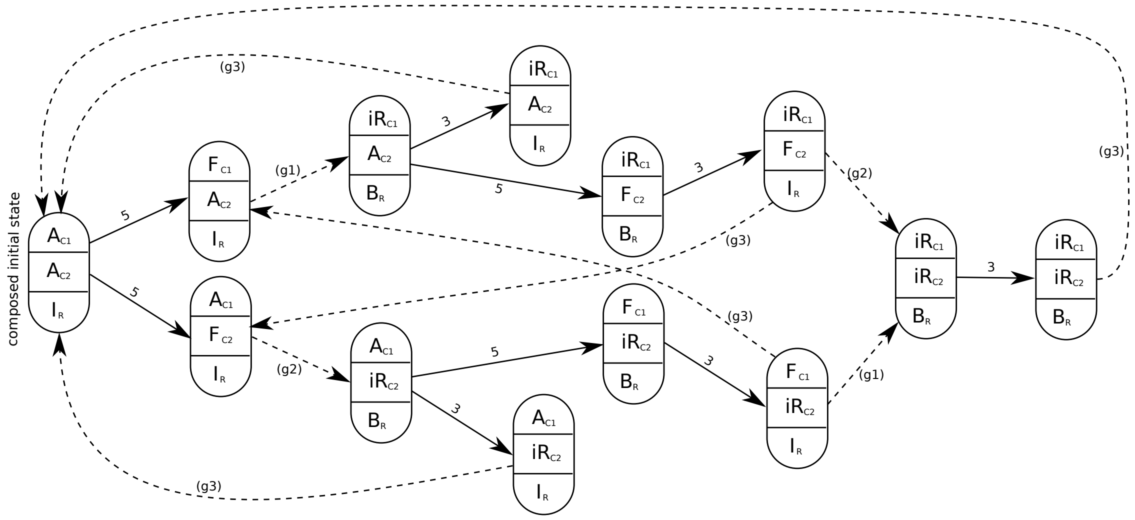


Figure 3.16: Example: Evolved state space

generative expression of a guards statement and the condition expression of a conditional reactive) and a reactive expression (contained by the conditional reactive):

$$CR' = [\,(\underbrace{grd.g \wedge c}_{=:g'}, r, grd.ns)^{x*y} \mid grd \in^x G_f \wedge (c,r) \in^y grd.CR]$$

Let $pt : \mathfrak{BE} \to \mathcal{P}(\mathfrak{BE} \times \mathbb{N})$ be a function which determines and enumerates the product terms. Furthermore, let $pt_{minterm} : \mathfrak{BE} \to \mathcal{P}(\mathfrak{BE} \times \mathbb{N})$ be a function which determines and enumerates the minterms of a Boolean expression. For a more detailed description of these functions see Appendix D. Additionally, let an enumeration function

$$enum : mset(\mathfrak{CE} \times \mathfrak{RE} \times \mathcal{NS}) \to \mathcal{P}(\mathfrak{CE} \times \mathfrak{RE} \times \mathcal{NS} \times \mathbb{N})$$

be given which enumerates all objects inside the multiset argument.

CR' can now be converted to a set CR_{pt} of encoded product term combinations:

$$\begin{aligned} CR_{pt} = \{\, & (pt_g, pt_r, \overbrace{(e_{ns}, e_{grd}, e_g, e_r)}^{enc \in \mathfrak{U}_{id}}) \quad \mid \\ & (pt_g, e_g) \in pt(c) \;\wedge\; (pt_r, e_r) \in pt_{minterm}(r) \;\wedge\; (c, r, e_{ns}, e_{grd}) \in enum(CR') \\ \} & \end{aligned}$$

The subsequent section will use the encoded product term combinations in order to derive the jointly acting transitions by taking choices among them into account.

Dealing with Competing Guarded Transitions

Let the set $\{A.\langle a_1\rangle, A.\langle a_2\rangle, B.\langle b\rangle\}$ represent a reactive minterm which is considered once state (S_A, S_B, S_C) will be reached. This is illustrated by Figure 3.17. Note that a guard label $\langle \mathtt{c} \rangle$ of instance C is used by two (competing) transitions. Furthermore, the instance A has two (competing) transitions which both reveal distinct labels $\langle \mathtt{a}_1 \rangle$ and $\langle \mathtt{a}_2 \rangle$. In general, an arbitrary number of competing transitions might be triggered by a minterm which refers to guard labels that belong to a single instance.

The semantics which determines how to deal with such a situation is not obvious. Several kinds of semantics could be defined:

1. A minterm will only be satisfied if it solely refers to a single label within a Behavior instance.

2. A minterm will only be satisfied if the addressed guard labels are enabled in terms of available guarded transitions. In case of satisfaction, all addressed competing transitions of a single Behavior instance are *in choice*.

3. A minterm will only be satisfied if all involved local transitions lead to the same state, such that they can be performed synchronously.

All above presented semantics provide advantages and disadvantages:

- When considering the $\wedge$ operator within the logical expression as a synchronisation of available behaviours, the example as given by the reactive minterm $\{A.\langle a_1\rangle, A.\langle a_2\rangle, B.\langle b\rangle\}$ will never be satisfied if one takes 1. as underlying semantics. Even though two labels are addressed hereby, it might also be the case that e.g. $C.\langle \mathtt{c} \rangle$ is addressed. This does not prevent from addressing competing transitions. It would require adding further restrictions. Anyway, it is counterintuitive to synchronise transitions of a single component which reveal themselves to be *in choice*. Apart from that, it is as well not a valid option to apply this to all possible cases.

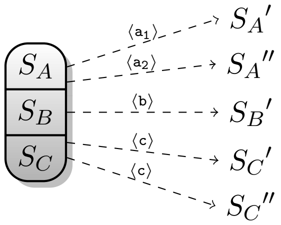

$\langle \mathtt{a}_1 \rangle$
$S_A{}'$
S_A
$\langle \mathtt{a}_2 \rangle$
$S_A{}''$
S_B
$\langle \mathtt{b} \rangle$
$S_B{}'$
S_C
$\langle \mathtt{c} \rangle$
$S_C{}'$
$\langle \mathtt{c} \rangle$
$S_C{}''$

Figure 3.17: Available guarded transitions within current state

- When considering 2., the minterm will be satisfied for $A.\langle a_1\rangle, A.\langle a_2\rangle$ if the availability of the referred guard labels is considered. This means that the $\wedge$ operator is not equivalent to a synchronisation because of the operator's nature.
- Definition 3. assumes that the $\wedge$ operator serves as a synchronisation and reasons about the availability of the referred label. The problem which will occur if this scheme is applied as the underlying semantics is that a local synchronisation between outgoing transitions has to be enforced. As described above, this only works for transitions that lead to the very same state. In most cases there will be no common state. This is why these cases have to be forbidden for the purpose of achieving an unambiguous target state.

For the implementation, the semantics given by 2. was chosen. This will be formalised in the following. A product term is hereby considered a set of literals, e.g. $l_1 \wedge l_2 \wedge \ldots \Leftrightarrow \{l_1, l_2, ...\}$ for reasons of simplification. This allows using the member operator $\in$. Let a tuple represent an encoded combination of product terms $(pt_g, pt_r, enc) \in^{>0} CR_{pt}$. Furthermore, let the composed state s and an index set $I^B = \{1, \ldots, n\}$ be defined (where n corresponds to the number of Behavior instances). The generative part pt_g of the tuple will be satisfied by s if

$$s \models pt_g \iff \quad \forall v \in pt_g \begin{cases} \nexists i \in I^B : v \triangleq s_i & \text{if } v \text{ negated} \\ \exists i \in I^B : v \triangleq s_i & \text{else.} \end{cases}$$

Note that $\triangleq$ means that the reference contained by the literal is compared to the addressed statement.

In order to determine whether the reactive part is satisfied, the outgoing guarded transitions for the current state s_i of the Behavior instances b_i have to be considered. For obtaining the set of outgoing (guarded) transitions of a given state s_i of a Behavior instance (component) b_i, the following function definition is given:

$$\rightrightarrows : (b_i, s_i) \mapsto [\, t^k \,|\, t \in^k (b_i.T \cap \mathfrak{T}_G) \wedge t.s \triangleq s_i \,]$$

For each component $i \leq n$, the sets of outgoing transitions can then be determined for a given state s_i:

$$(\underbrace{\rightrightarrows (b_1, s_1)}_{T_1^s}, \ldots, \underbrace{\rightrightarrows (b_n, s_n)}_{T_n^s})$$

The reactive part will be satisfied if no guarded transition can be found whose label corresponds to a negated literal v (when considering all negated literals v of a given reactive minterm) and if a guarded transition can be found for all unnegated literals:

$$s \vDash pt_r \iff \quad \forall v \in pt_r \begin{cases} \nexists i \in I^B : \exists t \in T_i^s : t \triangleq v & \text{if } v \text{ negated} \\ \exists i \in I^B : \exists t \in T_i^s : t \triangleq v & \text{else,} \end{cases}$$

For the purpose of determining the *set of enabled transitions* for each minterm (i.e. transitions that are available in the current composed state and referred to by the minterm literals of pt_r), a function is defined which considers only those transitions that are addressed by unnegated Boolean variables:

$$\rightrightarrows^{\surd} : T \mapsto [t^k \mid t \in^k T : \exists \text{ unnegated } v \in pt_r : t.l \triangleq v]$$

Whenever $\rightrightarrows^{\surd}$ is applied to a set of outgoing transitions of a component state, the set of enabled transitions for a single component is determined. It represents a local choice between possible behaviours of an instantiated Behavior with regard to a minterm pt_r. Accordingly, all choices are determined by the sets of enabled transitions $T_i^{\surd}$ for the current state $s \vDash pt_g \wedge s \vDash pt_r$:

$$(\underbrace{\rightrightarrows^{\surd} (T_i^s)}_{T_1^{\surd}}, \ldots, \underbrace{\rightrightarrows^{\surd} (T_n^s))}_{T_n^{\surd}})$$

Due to the chosen semantics 2., a number of combinations of jointly acting transitions arise among the addressed instances:

$$T_{combinations} = \prod_{i \in I_B : |T_i^{\surd}| > 0} T_i^{\surd} \tag{3.4}$$

Referring to the example given for Figure 3.17, the multisets of enabled transitions in the depicted composed state are given by

$$\left[[t_A^{\langle \mathsf{a}_1 \rangle}, t_A^{\langle \mathsf{a}_2 \rangle}], [t_B^{\langle \mathsf{b} \rangle}], []\right]$$

The choice in the first multiset leads to the combinatorics, determined by (3.4). The following competing tuples of jointly acting transitions are obtained which constitute a choice in the composed state space:

$$(S_A \to (S_A)', S_B \to (S_B)')$$
$$(S_A \to (S_A)'', S_B \to (S_B)')$$

The above can be generalised:

- The source states of a *composed transition* $T_c \in T_{combinations}$ are used to construct the successor s' by instantaneously switching the corresponding component state s_i of the current composed state s to the target states s'_i. This is addressed by the notation $s' = \langle T_c, s \rangle$ introduced on page 94.
- LARES provides two kinds of transition types, i.e. delayed (an exponential distribution) and weighted immediate transitions. It is by definition forbidden to derive composed distributions from heterogeneously typed distributions. A model will be considered *invalid* if a synchronisation between different types of distributions is defined. Let $(t_i)_{i \in I_B'} = T_c$. A property can be formulated which ensures that all transitions t_i comprise the same distribution type:

  $$\left| \bigcup_{t_i \in T_c} \pi_1(t_i.d) \right| \leq 1$$

  A guarded transition may also be specified without an explicit distribution given. Its distribution is then determined by the distribution type of the jointly acting transitions. If their distribution type is Markovian, the (neutral) rate 1 will be assumed or else if the distribution type is immediate, the (neutral) weight 1 will be assumed. If none of the addressed transitions provides a defined distribution, the immediate distribution with weight 1 will be taken as default for all the transitions. Based on the currently supported distribution types, the distribution of a composed transition is determined by either the weight w_c or the rate r_c:

  $$w_c = \prod_{t \in T_c} \pi_2(t.d) \qquad r_c = \prod_{t \in T_c} \pi_2(t.d) \tag{3.5}$$

Accordingly, *composed transitions* with an immediate distribution type or with an exponential distribution type are denoted by $s \overset{w_c}{\dashrightarrow} s'$ or $s \xrightarrow{r_c} s'$ respectively.

Completion of the LTS Semantic by SOS Rules for the Interaction Behaviour

The ideas described above can be used to define additional SOS rules which complement the rules that consider the unguarded behaviour. For both types of rules some common aspects can be stated beforehand:

- The **Side-conditions** ensure that all guards statements $g \in G_f$ are considered and transformed into tuples of generative/reactive product terms as introduced beforehand. Furthermore, the combinations of choices arising by virtue of multiple guard labels addressed within a single Behavior instance are constructed. For each choice a label and the unified distribution is determined.

- The **Antecedents** are fulfilled in case that a tuple of product terms $(pt_g, pt_r) \in CR_{pt}$ is satisfied by the current composed state s (i.e. considering the generative part $s \vDash pt_g$ and the reactive part $s \vDash pt_r$).
- As a **Consequence**, a composed state s leads to a composed state s' by each composed (labelled) transition which comprises the determined common distribution.

For each encoded tuple of generative/reactive product terms $(pt_g, pt_r, enc) \in CR_{pt}$ which is satisfied (cf. the antecedents of an SOS rules) as well as for each T_c a transition emerges:

- The following rule applies in case that a set of jointly acting transitions T_c yields an immediate distribution type:

$$\frac{s \vDash pt_g \quad \wedge \quad s \vDash pt_r}{s \overset{enc,w_c}{\dashrightarrow} s'} \quad \begin{smallmatrix} (pt_g,pt_r,enc)\in CR_{pt} \\ s'=\langle T_c,s\rangle \end{smallmatrix} \tag{3.6}$$

- The following rule applies in case that a set of jointly acting transitions T_c yields an exponential distribution type:

$$\frac{s \vDash pt_g \quad \wedge \quad s \vDash pt_r}{s \xrightarrow{enc,r_c} s'} \quad \begin{smallmatrix} (pt_g,pt_r,enc)\in CR_{pt} \\ s'=\langle T_c,s\rangle \end{smallmatrix} \tag{3.7}$$

Performing Reachability Analysis

Let s_1 be the composed initial state. The application of all SOS rules 3.2, 3.3, 3.6 and 3.7 on s_1 and recursively on all reachable composed successor states s_i' explores the overall set of reachable composed states. Once a fixed point is achieved (i.e., no further successive composed states can be reached that haven't been processed yet), the exploration ends. All transitions constructed by the rules are collected in the multiset T (such that $s \xrightarrow{r} s' \in^i T$ or $s \overset{w}{\dashrightarrow} s' \in^j T$) which represents the reachability graph. The resulting transitions system is denoted by the tuple (T, s_1). The resulting state space may contain different states which have both immediate and Markovian outgoing transitions. It is thus denoted as an *ESLTS* (Extended Stochastic Labelled Transition System). It is assumed that an immediate transition will always take place before a Markovian transition is able to perform, which is called the *maximum progress assumption*. Accordingly, all Markovian transitions within the set of outgoing transitions of a composed state which enables at least one immediate transition can be safely removed. In many cases an exhaustive analysis is required to be performed by tools that solely support SLTS. For this purpose, all vanishing states (whose sojourn time is 0 due to an outgoing immediate transition) have to be eliminated as described in [11].

3.5 From LARES$_{\text{BASE}}$ to Stochastic Process Algebra as Target Formalism

The process algebra that is used as a target formalism is the CASPA SPA. It will be introduced in the first part of this section. Besides, an intermediate formalism is presented which encapsulates sequential SPA processes and information stating how a process interacts with its environment (PACT – Process Algebra Composition Tuples). Complementary, its associated composition semantics is provided in order to calculate the set of synchronising action labels and to perform the composition according to the tree structure of the model. The final part of this section details the transformation from LARES$_{\text{BASE}}$ to PACT. It specifies how each instantiated Behavior is mapped to a sequential SPA process and determines how it interacts with its environment. The desired CASPA SPA specification ultimately results from folding the hierarchically structured PACT model.

3.5.1 Syntax and Semantics of the CASPA SPA

In order to specify the SPA semantics for the LARES language, the formal definition of the CASPA SPA (consisting of the syntactical notation and the semantics) is revisited. The most recent state of the process algebra tool CASPA is described in [11] and [123]. Previous publications that relate to CASPA can be found in [94, 119, 124, 125].

The set $\mathcal{L}$ of valid CASPA SPA expressions is defined by the following language elements. Let $Act = Act_M \dot{\cup} Act_I$ be the set of action names and Pro the set of process names. The action $\tau \in Act_I$ hereby denotes an internal, invisible activity. Furthermore, let $A_M \subseteq Act_M \times \Lambda$ be the set of Markovian actions (where $\Lambda = R^{\geq 0}$ is the set of rates), $A_I \subseteq Act_I \times W$ be the set of immediate actions (where $W = \mathbb{R}^{\geq 0}$ represents the set of weights) and $\sigma \in \mathcal{L}$ be the stop process. Let $P, Q \in \mathcal{L}$ denote two SPA processes, let $a \in Act$ represents an action label and let $X \in Pro$ be a process name. A new process definition can be specified by $X := P$ (hereby $:=$ assigns a process P to a given name X). A Markovian action prefix is denoted as $(a, r); P$ (where $r \in \Lambda$), an immediate action prefix is denoted as $(*a, w*); P$ (where $w \in W$), a choice is denoted as $P + Q$ and $P|[S]|Q$ is a parallel composition, where $S \subseteq Act \setminus \{\tau\}$ represents the set of labels of synchronising actions. Hereby, actions are only allowed to be synchronised in case they are of the same type (i.e. being either immediate or Markovian). The *hide* operator is denoted as $hide\,H\,in\,P$, within which $H \subseteq Act_I \setminus \{\tau\}$ consists of labels referring to actions which have to be hidden.

In the following, several SOS rules are given in order to formalise the semantics (of the action prefix, the choice operator, the parallel composition operator and hiding) for the

CASPA SPA by means of labelled transition systems, i.e. in terms of immediate and Markovian transitions:

- Semantic rules for an action prefix:

$$\frac{}{(a,w);P \overset{a,w}{\dashrightarrow} P} \qquad \frac{}{(a,\lambda);P \xrightarrow{a,\lambda} P}$$

- Semantic rules for the choice operator:

$$\frac{P \overset{a,w}{\dashrightarrow} P'}{P+Q \overset{a,w}{\dashrightarrow} P'} \qquad \frac{Q \overset{a,w}{\dashrightarrow} Q'}{P+Q \overset{a,w}{\dashrightarrow} Q'} \qquad \frac{P \xrightarrow{a,\lambda} P'}{P+Q \xrightarrow{a,\lambda} P'} \qquad \frac{Q \xrightarrow{a,\lambda} Q'}{P+Q \xrightarrow{a,\lambda} Q'}$$

- Parallel composition semantics
  - for the case that an action a is not in the synchronisation set:

$$\frac{P \xrightarrow{a,\lambda} P'}{P|[S]|Q \xrightarrow{a,\lambda} P'|[S]|Q}\ a \notin S \qquad \frac{P \overset{a,w}{\dashrightarrow} P'}{P|[S]|Q \overset{a,w}{\dashrightarrow} P'|[S]|Q}\ a \notin S$$

$$\frac{Q \xrightarrow{a,\mu} Q'}{P|[S]|Q \xrightarrow{a,\mu} P|[S]|Q'}\ a \notin S \qquad \frac{Q \overset{a,v}{\dashrightarrow} Q'}{P|[S]|Q \overset{a,v}{\dashrightarrow} P|[S]|Q'}\ a \notin S$$

  - for the case that an action a is in the synchronisation set:

$$\frac{P \xrightarrow{a,\lambda} P' \qquad Q \xrightarrow{a,\mu} Q'}{P|[S]|Q \xrightarrow{a,\lambda\cdot\mu} P'|[S]|Q'}\ a \in S \qquad \frac{P \overset{a,w}{\dashrightarrow} P' \qquad Q \overset{a,v}{\dashrightarrow} Q'}{P|[S]|Q \overset{a,w\cdot v}{\dashrightarrow} P'|[S]|Q'}\ a \in S \tag{3.8}$$

- The hide operator ensures that a certain action set is hidden (from an external point of view it is considered an internal action τ). An internal action cannot be observed and thus not be used for synchronisation with other processes. In the course of extending the CASPA language by immediate actions, hiding of Markovian actions has been disabled, such that only the following rules apply:

$$\frac{P \overset{a,w}{\dashrightarrow} P'}{\text{hide } H \text{ in } P \overset{\tau,w}{\dashrightarrow} \text{hide } H \text{ in } P'}\ a \in H \qquad \frac{P \overset{a,w}{\dashrightarrow} P'}{\text{hide } H \text{ in } P \overset{a,w}{\dashrightarrow} \text{hide } H \text{ in } P'}\ a \notin H$$

As it is described in [11], the arising transition system of a process algebra model is directly encoded by symbolic data structures and is subsequently converted to a purely Markovian model. However, the concrete syntax of CASPA comprises additional features which have not been discussed there:

- Parameters: A process definition can be parametrised, such that several parameters, each with a specific range of values in the domain of natural numbers, can be declared.
- Guards: Each subprocess defined within a parametrised process definition is guarded by constraints on the parameters, such that a subprocess is only able to perform when its guarding constraints are satisfied. An asterisk within a guard represents a wildcard which means that the associated subprocess is always able to act.

The following example specifies a simple *parametrised* queueing process P (with only a single parameter x representing the queue length):

```
P(x [2]) :=
  [x<2] (inkr,1); P(x+1)
  [x=2] (*full,1*); P(x)
  [x>0] (dekr,1); P(x-1)
  [x=0] (*empty,1*); P(x)
  [*] (doSth,1); P(x)
```

An equivalent *unparametrised* specification can be defined by

```
P0:= (*empty,1*);P0 + (inkr,1); P1 + (doSth,1); P0
P1:= (inkr,1); P2 + (dekr,1); P0 + (doSth,1); P1
P2:= (dekr,1); P1 + (*full,1*); P2 + (doSth,1); P2
```

Generally speaking, the cross-product over the parameter domains spans the potential state space of the sequential process. A choice between several guarded subprocesses arises when they can act alternatively (which is the case when their guarding constraints are satisfied due to the current parameter configuration). The introduction of parameters can thus be seen as an abbreviation notation in order to not enforce modellers to encode each explicit state in a sequential process by an individual process name. The EBNF grammar rules for the definition of the CASPA syntax including the above-mentioned parametrised processes are detailed in [123]. A classification of CASPA models is thereby established (to deal with compositionality issues as discussed in the beginning of the next section) with the purpose of giving a sufficient criterion for a model to be considered *nice*.

3.5.2 Compositionality Issues

The CASPA SPA (and hence the LARES approach defined thereupon) may be considered non-compositional in a strong sense. When specifying a process with a weight ratio of $1:1$ between two actions a and b in a choice, this ratio is naturally expected to be preserved in the final composed transition system by the probabilities $\frac{1}{2}$ for each transition (cf. [119]). Due to its composition with another process whose action label b is included in the synchronisation set, this ratio property might no longer be preserved. As determined by the SOS rule (3.8) for synchronised compositions, the resulting weight is calculated by the product of the weights from both processes, leading to a probability ratio of $1:2$ in the final composed transition system, as depicted by Figure 3.18. This fact requires a modeller to pay attention when additional components are composed as this may commonly not preserve the ratio between the weights/rates in terms of the resulting probabilities due to synchronisation.

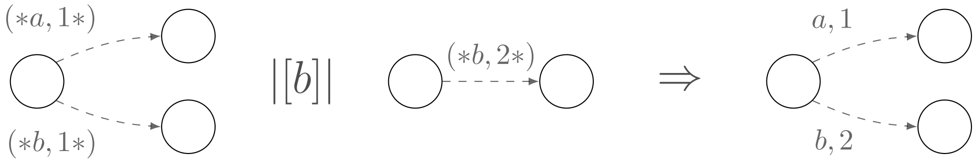


Figure 3.18: The ratio between two immediate actions is not preserved due to composition

The CASPA SPA as well as other formalisms which incorporate immediate transitions face another problem which derives from a fully interleaved parallel composition of a process with and another one without timeless traps. Whenever a timeless trap is reached by one process, the other process is not able to perform regarding the composed system (as shown in Figure 3.19). The reason is that time stops for the composed process, even though the composition defines no synchronisation and therefore should not cause any interference among the two processes with respect to the composed system. From a compositionality point of view this is counterintuitive because, in spite of one process entering a timeless trap, the other process should still be able to act. A solution for this compositionality issue may be based on stochastically discontinuous Markov processes as described in [7].

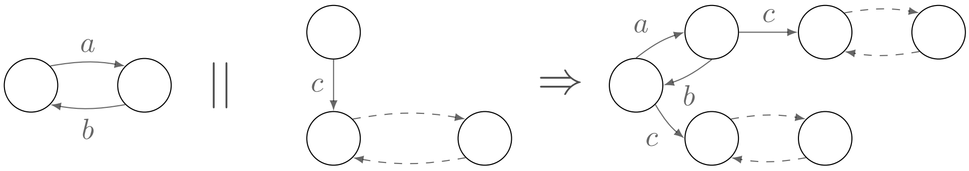


Figure 3.19: A process w/o timeless traps (on the very left) incurs a timeless trap

3.5.3 Intermediate Composition Structure of Process Algebra Terms

Since the CASPA SPA is a formalism that allows composing processes, it serves well to capture the hierarchy of a LARES model in terms of a composition structure. Each Module instance relates to a composition term, whereas a Behavior instance finally relates to a leaf node. The latter is expressed as a sequential SPA process which includes the required actions which originate from the guards statements of a LARES model in order to evaluate the assertion on states and perform the subsequent reaction.

This section introduces an intermediate representation to capture the generated sequential SPA processes, their interactions and the composition structure. It originally came into being in the course of development for transforming ZuverSicht models into an SPA [120].

The PACT (process algebra composition tuple) representation applies to other higher-level formalisms such as LARES as well. It encompasses, as its name suggests, several tuple definitions:

$$\mathcal{PACT} := \mathcal{PAT} \cup \mathcal{PACN}$$

Hereby, the set $\mathcal{PAT}$ (of process algebra tuples where each one represents a sequential process) and the set $\mathcal{PACN}$ (where each tuple represents a composition of several concurrent processes) are defined subsequently.

Let $\mathcal{S}$ denote the set of pieces of synchronisation information by which a process interacts with its environment. A tuple $(x, S, D) \in \mathcal{PAT}$ captures the (sequential) SPA process instantiation $x \in Pro$ which originates from a translated Behavior instance, the synchronisation information $S \in \mathcal{S}$ and the process definitions $D \in \mathcal{P}(\mathcal{L})$ which are associated with the instantiated process, hence

$$\mathcal{PAT} := Pro \times \mathcal{S} \times \mathcal{P}(\mathcal{L})$$

Instead of directly composing the above described tuples, the hierarchy is preserved by additional structures which are introduced to encompass the multiple subinstances of an instance. For this purpose, a (named) structure is defined to capture n-ary compositions of $\mathcal{PACT}$ elements

$$\mathcal{PACN} := Pro \times \mathcal{P}(\mathcal{PACT})$$

where $(x, C) \in \mathcal{PACN}$ provides an SPA process instantiation $x \in Pro$ subsuming a number of $\mathcal{PACT}$ elements C which are composed in a subsequent step.

While the CASPA SPA solely provides a binary composition operator, a $\mathcal{PACN}$ structure represents an n-ary composition of processes which has to be translated to a binary composed process term t. For this purpose, a representation is defined which temporarily captures these binary structures:

$$(t, S, D) \in \mathcal{PAC}, \text{ where } \mathcal{PAC} := \mathcal{L} \times \mathcal{S} \times \mathcal{P}(\mathcal{L})$$

Accordingly, it holds that $\mathcal{PAT} \subset \mathcal{PAC}$. $\mathcal{PAT}$ structures are distinguished as they represent atomic leaf elements of a process instance structure. However, as they are a subset of $\mathcal{PAC}$, the subsequently defined rules for $\mathcal{PAC}$ apply. The given (generic) structures can now be used to define the composition semantics thereof.

Roughly speaking, the PACT semantics allows synchronising among a set of actions given as part of the synchronisation information of two $\mathcal{PACT}$ structures. This results in a $\mathcal{PAT}$ structure that composes these two $\mathcal{PACT}$ structures and merges their synchronisation information.

The transformation from LARES$_{\text{BASE}}$ to SPA does not make use of the full capabilities defined for the PACT representation. The reason is, as initially mentioned, that PACT has not originally been devoted to LARES. Instead, each arising action from the guarded transitions inside the Behavior instances is treated with equal rights. According to that, the sets of actions which can synchronise grow along the instance paths towards the root instance, unless they are hidden. This depends on some specific implementation mechanism (which is not detailed here).

In order to illustrate the part of the PACT semantics which is applied for LARES, let the synchronisation information be a set of action labels (which synchronise with the environment of a single process) such that $\mathcal{S} := \mathcal{P}(Act)$. The semantic rule to compose two $\mathcal{PAC}$ structures (and implicitly, by their successive application, to compose a number of processes recursively) is given by:

$$\frac{(p, S_p, D_p) \qquad (q, S_q, D_q)}{\Big(\underbrace{(p|[S_p \cap S_q]|q)}_{\in\mathcal{L}}, \underbrace{S_p \cup S_q}_{\in\mathcal{S}}, \underbrace{D_p \cup D_q}_{\in\mathcal{P}(\mathcal{L})}\Big)} \tag{3.9}$$

In case a $\mathcal{PACN}$ structure (x, C) is processed, the following SOS rule can be applied to compose the associated $\mathcal{PACT}$ structures C. The application of this rule yields a new process definition $x := t_C$ which is a binary composition of all subprocesses obtained by the transformation of C (which in turn recursively depends on the application of the rules (3.9) and (3.10) and whose name x can be used for instantiation):

$$\frac{(x, C)}{\big(x, S_C, D_C \cup \{\underbrace{x := t_C}_{\text{process definition}}\}\big)} \tag{3.10}$$

When considering a LARES System instance which has been transformed into a $\mathcal{PACT}$ structure, x finally represents the instance name of the System and $D_C \cup \{x := t_C\}$ consists of all process definitions which encompass the full SPA specification of the model.

A more detailed explanation on the synchronisation aspects of the PACT structures can be consulted in Appendix C.

3.5.4 LARES$_{\text{BASE}}$ to SPA

In this section, the transformation from LARES$_{\text{BASE}}$ into SPA is detailed. As an introductory example let P, Q and T denote SPA processes.

A condition $P \wedge Q$ will imply a reaction in a system (composed of these processes) in case that a composed state may simultaneously behave like P and Q. In order to do so, a label l is introduced which identifies the above conjunctive expression. Furthermore, the first process needs to be able to perform an action with label l when it behaves like P, just like the other process when it behaves like Q. If both processes are composed in parallel including the action label as part of the synchronisation set S, an action will only take place in the composed state space if both processes can synchronously perform the action l. In this way, the rule (3.8) is applied to assert the above condition).

The following SOS rule considers T as a tester process $T \overset{l}{\dashrightarrow} T'$ which is composed with the other two processes. The process T hereby testifies condition $P \wedge Q$ to be satisfied by reaching T':

$$\frac{P \overset{l}{\dashrightarrow} P \qquad Q \overset{l}{\dashrightarrow} Q \qquad T \overset{l}{\dashrightarrow} T'}{P|[\mathcal{S}]|Q|[\mathcal{S}]|T \overset{l}{\dashrightarrow} P|[\mathcal{S}]|Q|[\mathcal{S}]|T'} \qquad l \in \mathcal{S}$$

To recall, a guards statement of a LARES$_{\text{BASE}}$ model depends on the state of its related instance tree (i.e. the condition expression) and triggers certain guard labels within the Behavior instances (by its reactive expression). Both condition and reactive expressions are Boolean expressions, similar to the condition given in the preceding example. The action l which relates to the given conjunctive expression is used inside the composition operator to realise the conjunction operator. Performing the action l in the composed state space is equivalent to the evaluation of the condition.

In general, action labels can be used to mimic conjunctive clauses (such as product terms). When a Boolean expression is translated to its sum of products, i.e. a disjunction of conjunctive clauses, each product term can be assigned with an individual label.

The following enumeration sketches the idea how such labels can be encoded for guards statements (which encompass arbitrary generative and reactive expressions):

1. Simply speaking, each guards statement describes interactions of process instances that have to be mapped within an SPA specification. This is achieved by synchronisation of action labels. For this purpose, each guards statement contributes a unique identifier to an action label (e.g. containing the namespace and the enumeration of the guards statement inside the module definition $Seq(\mathfrak{ID}) \times \mathbb{N}$).

2. The generative part of a guards statement is a Boolean expression which asserts on the current state of a number of instances. As above, it can be realised by parallel composition with defined synchronisation sets in order to mimic each product term of the DNF arising from the generative expression (cf. Appendix D). This also means that each product term has to contribute to an action label (e.g. by its enumeration).

3. The reactive part can be handled similarly to the generative part. Instead of asserting about the current states, it asserts about the currently possible (guarded) behaviour. If the reactive part did not imply a behavioural change, a timeless-trap would arise in the composed model in case of a synchronisation among immediate transitions. This is avoided by requiring at least one reference to a guard label to imply a behavioural change. In order to ensure that the intended semantics in terms of arising rates or weights in the composed state space is achieved, the combinations have to be built from the canonical DNF (also denoted as minterms) as reactive product terms (in contrast to the generative part, in which the simplified sum of product suffices). The reason for this is illustrated in the Appendix D.

Accordingly, an action label encoding which originates from a guards statement is determined by the namespace, the enumeration of a guards statement, the enumeration of a product term arising from the generative expression and the enumeration of a reactive product term arising from the reactive expression:

$$\mathcal{U}_{ID} = \mathcal{NS} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

Each literal inside a product term may address several Behavior instances. For the purpose of translating a LARES behaviour instance to an equivalent sequential SPA process (which incorporates the labels for the interaction among instances arising from guards statements), each transformation of a Behavior instance has to be supplied with the relevant literals of the product terms, their encodings and their distribution types. For each *product term combination* (consisting of product terms which originate from both the generative and the reactive part), the literals are augmented by the determined distribution type, the encoding and whether they come from a product term combination that only addresses a single behaviour instance. The latter is needed to subsequently exclude the arising action labels from the synchronisation information and therefore from the synchronisation set in order not to hinder the action from acting when being composed with other processes.

An augmented literal is composed of an encoding enc, a negated or unnegated literal v which refers to a Behavior instance (which reveals the specified guard label), a distribution type dt and an interaction flag f stating whether the literal's product term combination addresses multiple Behavior instances:

$$(enc, v, dt, f) \in \mathcal{AL}$$

The universal set of augmented literals is hence defined by

$$\mathcal{AL} = \mathcal{U}_{ID} \times (\{\neg, \not\neg\} \times \mathfrak{Ref}_R) \times D_{types} \times \mathbb{B}$$

Augmented literals are generated while traversing the model instances. For this purpose, the following traversal function is defined:

$$\tau_{spa} : \mathcal{I}_{\mathcal{M}} \times \mathfrak{FW}_{spa} \rightarrow \mathfrak{BW}_{spa}$$

As sketched beforehand, the augmented literals arising from the product term combinations have to be supplied to the transformation of the Behavior instances. For this purpose, the forward information is defined as follows:

$$\mathfrak{FW}_{spa} = Seq(\mathfrak{ID}) \times \mathcal{P}(\mathcal{AL}) \times \mathcal{P}(\mathcal{AL})$$

A piece of forward information $(ns, L_g, L_r) \in \mathfrak{FW}_{spa}$ is used to propagate the set of augmented literals L_g which contains the generative augmented literals and L_r which contains the reactive augmented literals towards the Behavior instances. Furthermore, the namespace ns is constructed whilst being propagated along the instance tree. The namespace will be used later to label the SPA process elements. The result of the transformation is a $\mathcal{PACT}$ structure:

$$\mathfrak{BW}_{spa} = \mathcal{PACT}$$

Forward Propagation

As previously mentioned, the transformation has to be supplied with the (augmented) literals coming from constructions of generative and reactive product terms. For each guards statement of a Module instance node, the generative and the reactive parts are transformed into their product term representations. Their combinations (i.e. the cross-product of the two sets of product terms) are uniquely encoded. For each product term combination, all literals are augmented by the mentioned attributes and are added to the forward information obtained from the environment. Subsequently, each augmented literal is forwarded towards the leaf node representing its addressed Behavior instance.

For the purpose of giving a formal description, the forward propagation function is defined such that its parameters (which encompass the instance i and the forward information fw) are mapped to a tuple whose forward information attribute fw' includes the information derived from fw and the current instance i:

$$\zeta_{spa} : (i, fw) \mapsto (i, fw')$$

The following paragraphs describe how the forward information fw is updated to fw'.

Augmented Literal Filtering. Only those literals are initially selected from the forward information fw that match the name of the processed instance i. For this purpose, the current namespace of the processed instance is determined by the obtained namespace $fw.ns$ and the name $i.l$ of the processed instance:

$$ns = fw.ns \circ i.l$$

All augmented literals $l \in fw.L_g$ and $l \in fw.L_r$ are subsequently compared whether their addressed namespace starts with the current namespace. For this purpose, the following notation is defined:

$$\underbrace{(x_1, \ldots, x_m)}_{\text{addressed namespace}} \text{ starts with } \underbrace{(y_1, \ldots, y_n)}_{\text{current namespace}} \iff n \leq m \wedge \forall_{i \leq n}\, x_i = y_i$$

The addressed namespace lns of an augmented literal l is derived by the concatenation $lns := \pi_1(l.enc) \circ \pi_2(l.v).i$ of the originating namespace and the addressed Behavior instance. As a result, all augmented literals which are relevant for the current subtree can be filtered:

$$L'_g = [\, l^k \mid l \in^k fw.L_g : lns \text{ starts with } ns\,] \qquad L'_r = [\, l^k \mid l \in^k fw.L_r : lns \text{ starts with } ns\,]$$

Figure 3.20 is a simplified depiction illustrating the propagation of augmented literals. It shows an instance tree which consists of the instances `m0`, `m1`, `m2`, `b0` and so forth, and a set of augmented literals originating from a guards statement of the root instance `m0`. Each tuple is hereby rendered incomplete and only consists of a literal of a distinct product term

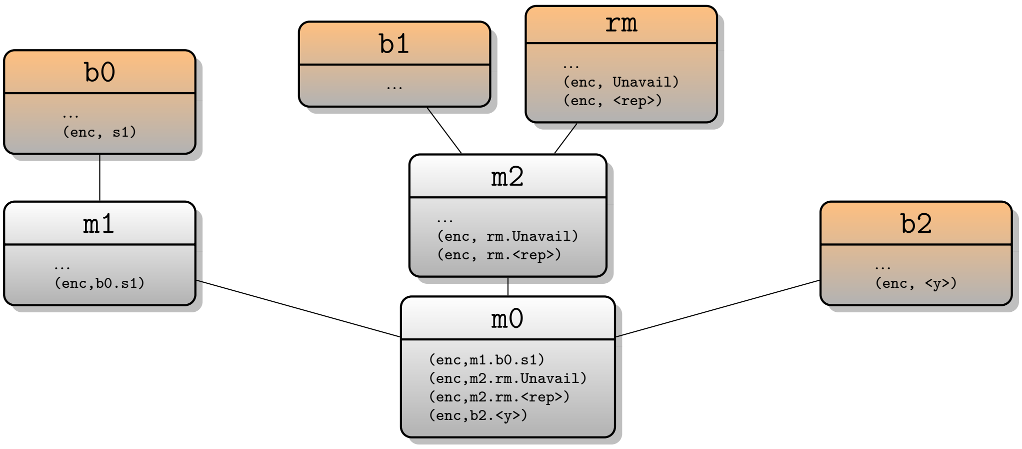


Figure 3.20: Filtering and propagation of augmented literals (rendered incomplete) into their matching subtrees using the successively adapted namespace of the literals' references

combination and the associated encoding `enc`. Each literal addresses a specific Behavior instance, traceable in the leaves of the subtrees. The namespace addressed by a literal which can be composed by `enc` and the addressed instance is compared to the namespace along the path to the Behavior instance in order to filter the relevant literals for a specific subtree and propagate them into it.

Augmented Literal Construction. As also described in the LTS transformation (cf. Section 3.4), each guards statement in the multiset of guards statements G (which is obtained from the body of the instance i, such that $G = i.t.b.G$) is composed of at least one *conditional reactive*. Correspondingly, for each *conditional reactive* a set of tuples is constructed. Each tuple contains a condition expression (built from the conjunction of the *generative condition* of the guards statement and the *restrictive condition* of the *conditional reactive*), a reactive expression (contained by the *conditional reactive*) and the namespace of a guards statement:

$$CR' = [\,(grd.g \wedge c, r, ns))^{x \cdot y} \mid grd \in^{x} G \wedge (c, r) \in^{y} grd.CR]$$

Let an enumeration function $enum : mset(\mathfrak{CE} \times \mathfrak{RE} \times \mathcal{NS}) \rightarrow \mathcal{P}(\mathfrak{CE} \times \mathfrak{RE} \times \mathcal{NS} \times \mathbb{N})$ be declared which enumerates all objects inside the multiset argument. Moreover, let the universal set of generative product terms be represented by $\mathfrak{M}_g := \mathcal{P}(\{\neg, \not\sqcap\} \times \mathfrak{Ref}_C)$ and the set of reactive product terms be described by $\mathfrak{M}_r := \mathcal{P}(\{\neg, \not\sqcap\} \times \mathfrak{Ref}_R)$.

The common distribution type is determined by using a function $\phi_{dt} : \mathfrak{M}_r \rightarrow D_{types}$. The set of distribution types of all considered literals in the reactive term mostly comprises a single distribution type dt. It may also be the case that none of the addressed guard labels define a distribution type. Therefore, the `immediate` distribution type is taken as default.

In case the set comprises two or more distribution types, the model is invalid as the CASPA semantics does not support an operator for merging heterogeneously typed distributions.

$$\phi_{dt} : pt_r \mapsto \begin{cases} \texttt{immediate} & \text{if } DT = \emptyset \\ dt & \text{if } DT = \{dt\} \end{cases}, \text{ where } DT := \{\, \underbrace{\pi_2(v).dt}_{\in D_{types}} \mid v \in pt_r \,\}$$

The set of product term combinations can be constructed and enumerated by using the functions $pt : \mathfrak{CE} \to \mathcal{P}(\mathfrak{M}_g \times \mathbb{N})$ and $pt_{minterm} : \mathfrak{RE} \to \mathcal{P}(\mathfrak{M}_r \times \mathbb{N})$ (whose definitions are outsourced to the Appendix D). As a result, the complete encoding can and the common distribution type can be derived for each combination:

$$\begin{aligned} &CR_{pt} = \{\, (pt_g, pt_r, \overbrace{(e_{ns}, e_{grd}, e_g, e_r)}^{\in \mathfrak{U}_{id}}, \phi_{dt}(pt_r)) \quad | \\ &\quad (pt_g, e_g) \in pt(c) \;\wedge\; (pt_r, e_r) \in pt_{minterm}(r) \;\wedge\; (c, r, e_{ns}, e_{grd}) \in enum(CR') \\ &\} \end{aligned}$$

Subsequently, the sets L_g and L_r are constructed consisting of augmented generative and reactive literals respectively. Both a generative literal and a reactive literal may trigger some behaviour without being synchronised with behaviours of other processes. It hence depends on the whole product term combination whether the Boolean interaction flag f is set to $true$ or $false$. The flag is obtained by a function $\phi_{dep} : \mathfrak{M}_g \times \mathfrak{M}_r \to \{true, false\}$ which determines whether all literals address a single Behavior instance:

$$\begin{gathered} L_r = \{\, (enc, l_r, dt, f) \mid l_r \in pt_r \wedge\; (pt_g, pt_r, enc, dt) \in CR_{pt}\} \\ L_g = \{\, (enc, l_g, dt, f) \mid l_g \in pt_g \wedge\; (pt_g, pt_r, enc, dt) \in CR_{pt}\} \\ \text{where } f = \phi_{dep}(pt_g, pt_r) \end{gathered}$$

Finally, the updated forward information fw' is composed of the new namespace and the sets of generative and reactive augmented literals (such that the new literals which have emerged due to the processing of the current instance are added to those coming from fw):

$$fw' = (ns, L'_g \cup L_g, L'_r \cup L_r)$$

Backward Aggregation

It now remains to define how all augmented literals which reach a specific Behavior instance are used to construct the actions within a local behaviour for interaction among the instances. For this purpose, the backward function is defined such that the Behavior instances of a given instance i are transformed into $\mathcal{PAT}$ structures taking the forward information $fw \in \mathfrak{FW}_{spa}$ into account in order to be finally transformed into a $\mathcal{PACT}$ instance i':

$$\beta_{spa} : (i, fw, BW) \mapsto \underbrace{i'}_{\in \mathcal{PACT}}$$

The set of Behavior instances $I_\mathcal{B}$ is obtained from the delegates inside the Module definition belonging to instance i (such that $I_\mathcal{B} = i.t.d$). The transformation function $t_\mathcal{B}$ of a Behavior instance has to be supplied with augmented literals inside the forward information fw. It is hence declared by $t_\mathcal{B} : \mathcal{I}_\mathcal{B} \times \mathfrak{FW}_{spa} \rightarrow \mathcal{PAT}$. As a result, the sequential SPA processes and their synchronisation information are obtained in terms of $\mathcal{PAT}$ elements:

$$I_\mathcal{B}^{\mathcal{PAT}} := \{\, t_\mathcal{B}(i_\mathcal{B}, fw) \quad | \quad i_\mathcal{B} \in I_\mathcal{B} \,\}$$

Let $t_\mathcal{B} : (i_\mathcal{B}, fw) \mapsto i'_\mathcal{B}$ define the transformation of a Behavior instance $i_\mathcal{B}$ to its $\mathcal{PAT}$ structure equivalent and let $fw = (ns, L_g, L_r)$ represent the forward information. The sequential SPA process and its interaction information are therewith derived and composed by means of a $\mathcal{PAT}$ structure.

Let L_g^{enc} and L_r^{enc} be two sets of augmented (generative/reactive) literals which have the same encoding enc and which correspond to a specific Behavior instance $i_\mathcal{B}$, such that the namespace $ns_\mathcal{B} := fw.ns \circ i_\mathcal{B}.l$ of the Behavior instance matches a literal's namespace:

$$\begin{aligned} L_g^{enc} &= \{\, l_g \in L_g \mid l_g.enc = enc \wedge ns_\mathcal{B} = l_g.ns \circ \pi_2(l_g.v).i \} \\ L_r^{enc} &= \{\, l_r \in L_r \mid l_r.enc = enc \wedge ns_\mathcal{B} = l_r.ns \circ \underbrace{\pi_2(l_r.v)}_{\in \mathfrak{Ref}_R}.i \} \end{aligned}$$

The set of generative augmented tuples L_g^{enc} and the set of reactive augmented tuples L_r^{enc} can be partitioned using the criterion whether the contained literal is negated or unnegated:

$$\begin{aligned} L_g^{\not\neg} &= \{l \in L_g^{enc} \,|\, \pi_1(l.v) = \not\neg\} \qquad & L_g^{\neg} &= \{l \in L_g^{enc} \,|\, \pi_1(l.v) = \neg\} \\ L_r^{\not\neg} &= \{l \in L_r^{enc} \,|\, \pi_1(l.v) = \not\neg\} \qquad & L_r^{\neg} &= \{l \in L_r^{enc} \,|\, \pi_1(l.v) = \neg\} \end{aligned}$$

Based on the above classes, several cases for the transformation of a Behavior instance have to be distinguished. The following part formalises the construction of actions (for a single encoding enc) which are used to carry out the interactions among sequential processes.

Let the set of states of the processed Behavior instance be $S := i_\mathcal{B}.t.b.S$. The states in the set S are filtered with regard to L_r (i.e. the reactive product term part of a specific combination that addresses the single Behavior instance $i_\mathcal{B}$). The set of reactive augmented literals L_r will be satisfied by a state $s \in S$ if for all non-negated literals there exists a guarded transition t which starts in s such that its guard label is referred to by the literal's variable, and if there is no guarded transition which corresponds to a negated literal:

$$s \vDash L_r :\Longleftrightarrow \Big(\forall l \in L_r^{\not\neg} \exists t \in \delta_g(s) \,:\, \pi_2(l.v).r = t.l\Big) \wedge \Big(\forall l \in L_r^{\neg} \nexists t \in \delta_g(s) \,:\, \pi_2(l.v).r = t.l\Big)$$

The generative literals can be used in order to provide a similar notation to assert whether a state s satisfies L_g (representing the generative condition part of a product term combination). This will be the case if all non-negated literals refer to the current state s and all negated literals do not refer to s:

$$s \vDash L_g :\Longleftrightarrow \Big(\forall l \in L_g^{\not\neg} \,:\, s = \pi_2(l.v).r\Big) \wedge \Big(\forall l \in L_g^{\neg} \,:\, s \neq \pi_2(l.v).r\Big)$$

If a state satisfies the non-empty local part $L_g \cup L_r$ of an original product term combination as the fraction of literals which reach and therefore address the currently processed Behavior instance (i.e. $s \vDash L_g \wedge s \vDash L_r$), it depends on the common distribution type whether the constructed transition will be immediately or exponentially typed. Let $dt : \mathcal{AL} \to \mathcal{D}_{types}$ declare a function which extracts the common distribution type of the local part of augmented literals which belong to a given product term combination. The common distribution type of a product term combination will always equal the distribution type of an addressed guarded transition if a distribution type is defined for the guarded transition t (such that $t.d \neq ()$). For this purpose, only the rate or weight value has to be determined from a given transition using a function $dv : \mathcal{T} \to \mathbb{R}$ which is defined as

$$dv : t \mapsto \begin{cases} \pi_2(t.d) & \text{if } t.d \neq () \\ 1.0 & \text{if } t.d = () \end{cases}$$

The following rule can be assigned whenever a state s satisfies the non-empty local part of a product term combination among others by the virtue of unnegated reactive literals (indicated by the non-empty set $L_r^{\not\neg}$):

$$\frac{s \vDash L_g \wedge s \vDash L_r}{\underbrace{s \overset{l.enc}{\rightsquigarrow}_d t.t \,+\, \ldots \,+\, s \overset{l.enc}{\rightsquigarrow}_d t.t}_{k \text{ times}}} \quad \begin{matrix} \exists l{\in}L_r^{\not\neg}\ \exists t{\in}^k \delta_g(s) : t.l{=}\pi_2(l.v).r \\ d{=}(dt(L_g{\cup}L_r),dv(t)) \end{matrix} \tag{3.11}$$

Hereby, each addressed guarded transition t will yield an action prefix (denoted as $\rightsquigarrow$) of a follow-up behaviour which is determined by the projection of the target state $t.t$ of the transition. The multiplicity k of each guarded transition is considered by a choice among the arising terms. Each action prefix is composed of a label enc and a distribution d. The distribution d is determined by the common distribution type of the related product term combination (using the function dt) and by a value which represents a weight (in case of an immediate distribution type) or a rate (in case of an exponential distribution type) which will be 1.0 as default if no explicit distribution has been defined for t.

When state s satisfies the non-empty local part of a product term combination solely by the virtue of the negated literals (indicated by $L_r^{\neq} = \emptyset \ \wedge \ L_g \cup L_r^{\neg} \neq \emptyset$), the following rule applies, where the distribution d is given for this case by the common distribution type and the default value 1.0:

$$\frac{s \models L_g \wedge s \models L_r}{s \overset{l.enc}{\rightsquigarrow}_d s} \quad {\scriptstyle L_r^{\neq}=\emptyset \,\wedge\, L_g \cup L_r^{\neg} \neq \emptyset \atop d=(dt(L_g \cup L_r),1.0)} \tag{3.12}$$

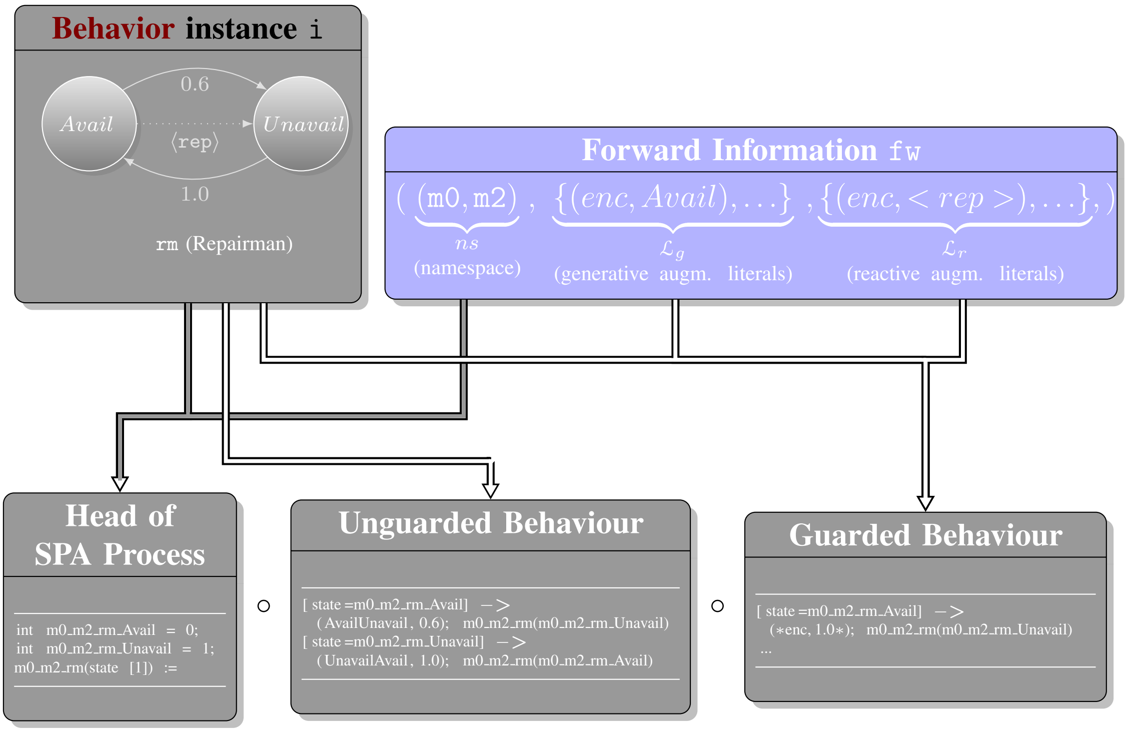
Let $U_{ID}^{i_{\mathcal{B}}}$ be the set of all encodings enc which occur during the processing of a Behavior instance $i_{\mathcal{B}}$. Let all transitions which are generated for a specific encoding enc due to the application of the above rules on all states $s \in S$ be aggregated within a multiset T^{enc}. The union of all these multisets is given by $T^{i_{\mathcal{B}}}$:

$$T^{i_{\mathcal{B}}} = \bigcup_{enc \in U_{ID}^{i_{\mathcal{B}}}} T^{enc}$$

Let $T_u^{i_{\mathcal{B}}} = \{\, \delta_u(s) \mid s \in S\}$ unify the unguarded transitions of $i_{\mathcal{B}}$. Hereby, δ_u extracts all outgoing unguarded transitions for each state and transforms them into the form $s \overset{sot}{\rightsquigarrow}_d t$ within which a transition is labelled by the concatenation of its source and target state label. As a result of this, the sequential SPA process of the Behavior instance can be derived. Therefore, the process name has to be constructed (which is built upon the namespace ns and the behaviour label $i_b.l$). A function $ns2spa : \mathcal{NS} \rightarrow Pro$ (whose definition is not formally detailed) is used to map a LARES namespace to a valid CASPA SPA process name $P_{name} = ns2spa(ns \circ i_b.l)$. Let the function $nc : \mathcal{S} \rightarrow \mathbb{N}$ be declared which maps each state $s_0, \ldots, s_{n-1}$ in the set of Behavior states S to its subscript value i ($0 \leq i < n$). Furthermore, a process parameter `state` spanned by the above enumeration is declared. The function Γ for transforming a transition into an action prefix can then be defined:

$$\Gamma : s \overset{l}{\rightsquigarrow}_d s' \mapsto \begin{cases} \texttt{[state=}nc(s)\texttt{]-> (*}l\texttt{,}w\texttt{*);}P_{name}\texttt{(}nc(s')\texttt{)} & \text{if } d = (\texttt{immediate}, w) \\ \texttt{[state=}nc(s)\texttt{]-> (}l\texttt{,}r\texttt{);}P_{name}\texttt{(}nc(s')\texttt{)} & \text{if } d = (\texttt{exponential}, r) \end{cases}$$

The guarded behaviour $T^{i_{\mathcal{B}}}$ and the unguarded behaviour $T_u^{i_{\mathcal{B}}}$ can be used to derive the overall sequential process. For this purpose, let all transitions be represented as $[t_1, t_2, \ldots] = T^{i_{\mathcal{B}}} \cup T_u^{i_{\mathcal{B}}}$. The sequential process for $i_{\mathcal{B}}$ can then be derived by taking the process name, the process parameter definition and the process body into account. The latter is constructed by the concatenation of all guarded processes which originate from all transitions $[t_1, t_2, \ldots]$: $P_{name}\underbrace{\texttt{(state [}|n|\texttt{])}}_{\text{parameter definition}} := \underbrace{\Gamma(t_1) \circ \Gamma(t_2) \circ \ldots}_{\text{guarded processes}}.$


Behavior instance i
0.6
Avail
Unavail
⟨rep⟩
1.0
rm (Repairman)
Forward Information fw
$(\underbrace{(\texttt{m0,m2})}_{ns}, \underbrace{\{(enc, Avail), \ldots\}}_{\mathcal{L}_g}, \underbrace{\{(enc, <rep>), \ldots\}}_{\mathcal{L}_r})$
(namespace)
(generative augm. literals)
(reactive augm. literals)
Head of SPA Process
○
Unguarded Behaviour
○
Guarded Behaviour

```
int  m0_m2_rm_Avail  =  0;
int  m0_m2_rm_Unavail  =  1;
m0_m2_rm(state  [1])  :=
```

```
[ state =m0_m2_rm_Avail]  −>
  ( AvailUnavail , 0.6);   m0_m2_rm(m0_m2_rm_Unavail)
[ state =m0_m2_rm_Unavail]  −>
  ( UnavailAvail , 1.0);   m0_m2_rm(m0_m2_rm_Avail)
```

```
[ state =m0_m2_rm_Avail]  −>
  (∗enc, 1.0∗);   m0_m2_rm(m0_m2_rm_Unavail)
  ...
```

Figure 3.21: Translation of a Behavior instance into an SPA Process

In Figure 3.21 the Behavior instance translation is depicted. It shows the arguments `i` and `fw` of the backwards method which are defined for this example by a repair behaviour `rm`, taken from Figure 1.13(c), and by some forward information which relates to Figure 3.20. It illustrates which parts are used in order to construct the sequential SPA process thereof.

Let p denote a constructed process definition. The $\mathcal{PAT}$ structure $i'_{\mathbb{B}} = (P_{name}(init), s, \{p\})$ is then composed from the process name P_{name}, the initial state $init$ which is derived from $i_{\mathbb{B}}.t.ic.l$, the process definition p and the synchronisation information s encompassing all encodings enc which have been processed and whose corresponding augmented literals were not interacting with other components indicated by the interaction flag f (which is $false$ the case that solely $i_{\mathbb{B}}$ is involved in a generative/reactive product term combination).

All translated Behavior instances $I_{\mathbb{B}}^{\mathcal{PAT}}$ and all translated subinstances inside BW are subsequently composed in terms of a $\mathcal{PACN}$ structure which represents the PACT translation of a Module instance i including all translations of its substructures. For that purpose, the instance namespace ns is used as its name and all translated $\mathcal{PACT}$ substructures are unified:

$$i' = (ns2spa(ns), I_{\mathbb{B}}^{\mathcal{PAT}} \cup BW)$$

3.6 Assuring the Correctness of different Transformation-Semantics

It is difficult to show that the defined semantics and the implemented transformation are consistent with respect to the resulting behaviour. This requires checking that for all possible LARES input models the implementations of both the LTS transformation and the SPA transformation yield *behaviourally equivalent* models.

Two resulting labelled transition systems will be considered *behaviourally equivalent* if there is a bisimulation relation between them, which follows a desired bisimulation notion (see also [126]). Strong bisimulation and weak bisimulation can be distinguished (the latter extends strong bisimulation by a special treatment of the unobservable action τ). LARES models are, as described in the previous transformation sections, mapped to extended stochastic labelled transition systems, which allow Markovian and immediate transitions. More importantly, immediate transitions can be eliminated in a post-processing step, which makes a detailed definition of weak bisimulation unnecessary. Due to this reason, Section 3.6.1 describes how strong bisimulation is applied to check whether two systems (described using the LARES language) show an equivalent behaviour. This approach is based on testing and was initially implemented in order to find errors in the semantics and the transformation implementation by employing the diversification of code using redundant transformations (as both transformations are required to transform from the same input model into behaviourally equivalent output models). Although formally verifying the transformation implementation is beyond the scope of this work, Section 3.6.2 attempts to sketch a proof on the equivalence of the presented formal transformation semantics.

3.6.1 Testing Based Approach

The testing based approach uses a test-set of models on which the transformations are performed. The standard state-based algorithm introduced in [84] is applied to prove functional and Markovian equivalence between resulting transition systems. In order to stay consistent with the remaining work, the function $checkEquivalence$ checking the bisimulation equivalence of a number of transition systems as given in Algorithm 1 is based on the abstract representation $\mathcal{TRA}$ for labelled transition systems.

Let $\mathfrak{C} := \mathcal{P}(\mathcal{S})$ denote the universal set of classes where each class consists of a distinct set of states. For a class $c \in \mathfrak{C}$ and an index set $I \subset \mathbb{N}$, let $p_I^c = \{c_i \mid i \in I\}$ be a partition of c. Hence, for the non-empty sets $c_i \subseteq c$ it holds that $\bigcup_{i \in I} c_i = c$ and $\forall c_i, c_j \in p_I^c \,:\, c_i \cap c_j = \emptyset$ if $c_i \neq c_j$. $\mathfrak{S} := \Sigma^* \times \mathfrak{C}$ is the universal set of splitters, where each splitter comprises a transition label and a class.

Let $\mathfrak{R}$ denote the universal set of relations between a source state and its outgoing transitions. The algorithm requires two arguments. These are a function for the equivalence notion $\mathfrak{en}$ and a set of transition systems TS. The algorithm works on the union set S of distinct states of the given transition systems (cf. lines 1-4), in order to eventually build the initial partition S (cf. line 5), and constructs a set of splitters for each label of a Markovian transition by building the cross product of the set of labels and the initial partition (cf. lines 6 - 7). Let $R \in \mathfrak{R}$ (cf. line 8) be a relation, where $sR[s \rightarrow s', s \rightarrow s'', ...]$. As long as there are splitters available (cf. line 9), meaning that no fixed point has yet been reached, a splitter is taken from the set of splitters (cf. line 10) in order to refine the partition (cf. lines 11-18). The way how a partition is established is externalised by a function $split$. It is performed on each class c in the partition using the given splitter spl (cf. line 12). In the following (cf. lines 13-16), the obtained set of subclasses C' are added to the partition, whereas the original class c is removed. Furthermore, for each class in C' also new splitters are derived and added to the set of splitters, whereas the splitter related to the original class is removed. Irrespective of a further partition refinement, the current splitter spl has to be removed (cf. line 17). Once the fixed point is reached (i.e. no further splitters are available), the outmost loop will end. The transition systems are bisimulation equivalent (cf. line 20) in case that the equivalence formula evaluates that all initial states of the given transition systems are still contained by a single class of the refined final partition.

To refine a partition, each class c of the partition is subdivided corresponding to a criterion which will be evaluated for each contained state in c. Let $\mathcal{U}$ be a set of elements, which serve as the criterion for the construction of equivalence classes within the algorithm. A function $\mathfrak{en}$ can be declared which defines the notion of equivalence. For this purpose, each state is classified by taking its outgoing transitions and a given splitter into account:

$$\mathfrak{en} : mset(\mathcal{T}) \times \overbrace{\underbrace{\Sigma^*}_{\substack{\text{splitter}\\ \text{label}}} \times \underbrace{\mathfrak{C}}_{\substack{\text{splitter}\\ \text{class}}}}^{\mathfrak{S}:=}) \rightarrow \mathcal{U}$$

Different types of equivalence notions are given by defining the function $\mathfrak{en}$:

- Strong bisimulation (considering only the transition label, cf. [84]): A class c will be split into the partitions $c_\top$ and $c_\perp$, depending on the transitions emanating from the states $s \in c$ comprising the label a with destination into C_{split}

$$\mathfrak{en}_{\xrightarrow{a}} : (T, a, C_{split}) \mapsto \underbrace{\overbrace{\left\{ l \mid s \xrightarrow{l,\lambda} s' \in^k T \wedge s' \in C_{split} \wedge l = a \right\}}^{\text{either } \{a\} \text{ or } \emptyset} \neq \emptyset}_{\in \mathbb{B}}$$

Algorithm 1 checkEquivalence : $(\mathfrak{en}, TS)$

```
$T := \dot{\bigcup}_{ts \in TS} ts.T$
$S_{source} := \{s \,|\, s \rightarrow t \in T\}$
$S_{target} := \{t \,|\, s \rightarrow t \in T\}$
$S := S_{source} \cup S_{target}$
$Partition := \{S\}$
$L := \{l \,|\, s \overset{l,v}{\rightarrow} t \in T\}$
$Splitters = L \times Partition$
$R := \{\, (s, \{t \in T \,|\, s = t.s\}) \mid s \in S_{source} \,\}$
while $Splitters \neq \emptyset$ do
    take $spl \in Splitters$
    for all $c \in Partition$ do
        $C' := split_{\mathfrak{en}}(R, c, spl)$
        if $C' \neq \emptyset$ then
            $Partition := (Partition \cup C') \setminus \{c\}$
            $Splitters := (Splitters \cup (L \times C')) \setminus (L \times \{c\})$
        end if
        $Splitters := Splitters \setminus \{spl\}$
    end for
end while
return $\exists c \in Partition : \forall_{ts \in TS}\; ts.s_0 \in c$
```

- Markovian bisimulation (taking only the cumulative rate into account): A class c will be split into k subclasses $c_{r1}, c_{r2}, ..., c_{rk}$, depending on the cumulative rates calculated from E (i.e. the multiset of all transitions emanating from a state $s \in c$ with destination into C_{split})

$$\mathfrak{en}_{\xrightarrow{r}} : (T, a, C_{split}) \mapsto \underbrace{\sum_{\lambda \in^n E} \lambda \cdot n}_{\in \mathbb{R}}, \text{ where } E := \bigcup_{s' \in C_{split}} \left[\lambda^k \;\middle|\; s \xrightarrow{l,\lambda} s' \in^k T \right]$$

- Markovian strong bisimulation (considering both the transition label and their cumulative rate, cf. [84]): A class c will be split into k subclasses $c_{r1}, c_{r2}, ..., c_{rk}$, depending on a certain label a and the cumulative rates calculated from E (i.e. the multiset of all transitions with label a emanating from a state $s \in c$ with destination into C_{split})

$$\mathfrak{en}_{\xrightarrow{a,v}} : (T, a, C_{split}) \mapsto \underbrace{\sum_{\lambda \in^n E} \lambda \cdot n}_{\in \mathbb{R}}, \text{ where } E := \bigcup_{s' \in C_{split}} \left[\lambda^k \mid s \xrightarrow{l,\lambda} s' \in^k T \wedge l = a \right]$$

Within Algorithm 1 the function $split$, parametrised by the bisimulation notion $\mathfrak{en}$, is called to perform the partitioning of c into the equivalence subclasses C':

$$split_{\mathfrak{en}} : \mathfrak{R} \times \mathfrak{C} \times \mathfrak{S} \to \mathcal{P}(\mathfrak{C})$$

By constructing each sub-partition and by grouping each element in the partition, $split_{\mathfrak{en}}$ is defined in such a way that all states $s \in c$ are grouped corresponding to the result of the equivalence criterion function

$$split_{\mathfrak{en}} : (R, c, (l, C)) \mapsto c/_{\overset{\mathfrak{en}}{=}},$$

where for $s, t \in c$ the relation $\overset{\mathfrak{en}}{=}$ is defined by

$$s \overset{\mathfrak{en}}{=} t \quad \Longleftrightarrow \quad \mathfrak{en}(R(s), l, C)) = \mathfrak{en}(R(t), l, C))$$

Numerous test models have been developed in order to cover as many aspects of the language as possible. Beside standard meaningful test cases, further models have been constructed in order to capture also special cases of the language and the applied transformations. In doing so, a lot of semantic issues and programming errors have been fixed. Moreover, the set of test cases has grown over time and is continuously reused for checking the correctness of the transformation semantics and the transformation code.

3.6.2 General Proof Sketch

Checking bisimulation equivalence between the results of both transformations for a set of test models is not exhaustive and is thus not sufficient as a general proof which ensures that the transformations implement equivalent semantics by means of labelled transition systems. Taking the SOS-semantics for the CASPA stochastic process algebra into account, a proof has to generally show that the SPA transformation constructs models which are behaviourally equivalent to those obtained by the application of the LTS transformation rules of the LARES$_{\text{FLAT}}$ representation. The necessary steps are also illustrated by example (starting on p. 125).

Due to the complexity of the language this proof cannot detail each aspect of the transformation. However, starting from LARES$_{\text{BASE}}$ to apply the process algebra semantics and from LARES$_{\text{FLAT}}$ to apply the LTS rules should be sufficient, as the flattening of a LARES$_{\text{BASE}}$ model can be considered trivial.

The aim is to show the existence of a bisimulation relation $\sim$ between the results of the SPA transformation (1) including the SOS rules for the process algebra (2) and the LTS transformation (3) as depicted in Figure 3.22. An even stronger assumption (and thus sufficient for the required bisimulation equivalence proof) is to show that there is a bijective mapping between the states and between the transitions of two transition system $\check{S}$ and $\hat{S}$. Hereby, the number of transitions (with a specific labelling) $\check{s} \xrightarrow{l} \check{s}' \in \check{S} \times \mathcal{E} \times \check{S}$ has to coincide with the number of transitions (of equivalent labelling) $\hat{s} \xrightarrow{l} \hat{s}' \in \hat{S} \times \mathcal{E} \times \hat{S}$, where $\hat{s} \sim \check{s}$ and $\hat{s}' \sim \check{s}'$ (i.e. an isomorphism $\cong$ of labelled multigraphs).

$$\begin{tikzcd}
 & \text{LARES}_{\text{BASE}} \arrow[dl, "(1)"'] \arrow[dr, "\text{trivial}"] & \\
 & & \text{LARES}_{\text{FLAT}} \arrow[dd, "(3)"] \\
SPA \arrow[d, "(2)"] & & \\
LTS \arrow[rr, leftrightarrow, dotted, "(4)\ \cong"] & & LTS
\end{tikzcd}$$

Figure 3.22: Organisation of the proof sketch

Following the outline of Figure 3.22 the proof includes the following transformation steps:

(1) The SPA transformation rules are used to construct the sequential processes: Action prefixes arise due to unguarded transitions, or due to guarded transitions triggered by some reactive expression of a guards statement or due to self-loops to query a specific state required by a condition expression of a guards statement. The disjunctive generative/reactive product term representation is used, as satisfiability of a single generative/reactive product term can be mimicked by a set of synchronised actions (where each action can be performed when a Behavior's process satisfies its part of the product term). Satisfiability of a complete Boolean expression will thus be given if one of these sets of actions can be performed jointly. This also holds for the combination of generative and reactive product terms. All action prefixes which arise from the same product term combination are thus uniquely encoded. Each constructed process offers these action labels to its environment for synchronisation. A binary composition tree is built which captures the structure of a LARES model. Whenever two processes offer an action label, it is included in the synchronisation set.

(2) If a single process can perform an action prefix, this action may also occur in the LTS independent of a composition with other processes. Whenever two action prefixes of two distinct processes have the same action label (which is part of the synchronisation set of the processes' composition), a precondition is satisfied which allows an SOS rule to be applied in order to derive the corresponding LTS due to

the action's synchronised execution. An inductive application of the rule allows composing all processes which can perform equally labelled actions (associated with a single product term). The composition of all process terms corresponds to a composed state of the overall state space. Whenever equally labelled actions can be performed jointly, the composed term transitions into its successor term which represents the successor state.

(3) The LTS rules directly operate on a composed state (which is initially given by the initial state of the Behavior instances). Whenever a generative/reactive combination is satisfied, a successor state can be derived via a joint transition which brings all component states into their successor states (there might be different combinations due to choices within each Behavior instance).

The proof can be derived from the above transformation rules:

(4) It has to be shown that the composition and simplification of the rules of the transformation steps (1) and (2) always yield a transition system where each state represents a composed process term $\breve{s}$ which can bijectively be mapped (see p. 131) to a state $\hat{s}$ of the transition system which arises by taking the transformation path along step (3). Furthermore, it has to be ensured that the number of each (specifically attributed) transition starting in a state $\breve{s}$ and aiming at a composed successor term $\breve{s}'$ coincides with the number of (equivalently attributed) transitions to the related successor states $\hat{s}$ and $\hat{s}'$. In this case, the resulting state spaces are isomorphic.

Having outlined the idea of the proof, an example is drawn, before formalising the proof sketch, in order to illustrate the required steps and to raise the notion of behavioural equivalence by means of the property of isomorphism.

Illustration of the Proof Sketch by an Example

Figure 3.23 depicts a substructure of an instance tree which contains a guards statement inside some Module instance p. Additionally, the literal propagation (as performed during an SPA transformation) is illustrated at each edge of the instance tree (by boxes containing the propagated literals). Only a single product term combination arises thereof (since both the generative and a reactive expression are conjunctive terms), which leads to a number of augmented literals (i.e. tuples comprising a negated or an unnegated literal and the unique encoding `enc` determined for the combination). These augmented literals are propagated towards the addressed subinstances. Once the addressed Behavior instance is reached, this information will be used to construct the sequential SPA process by including the action prefixes as described by rules (3.11) and (3.12).

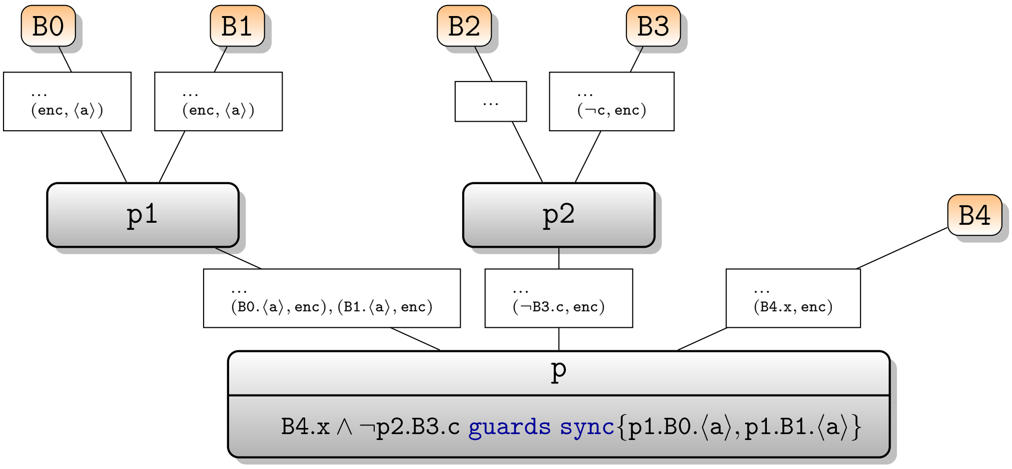


Figure 3.23: Encoded literal distribution for the SPA translation of Behavior instances

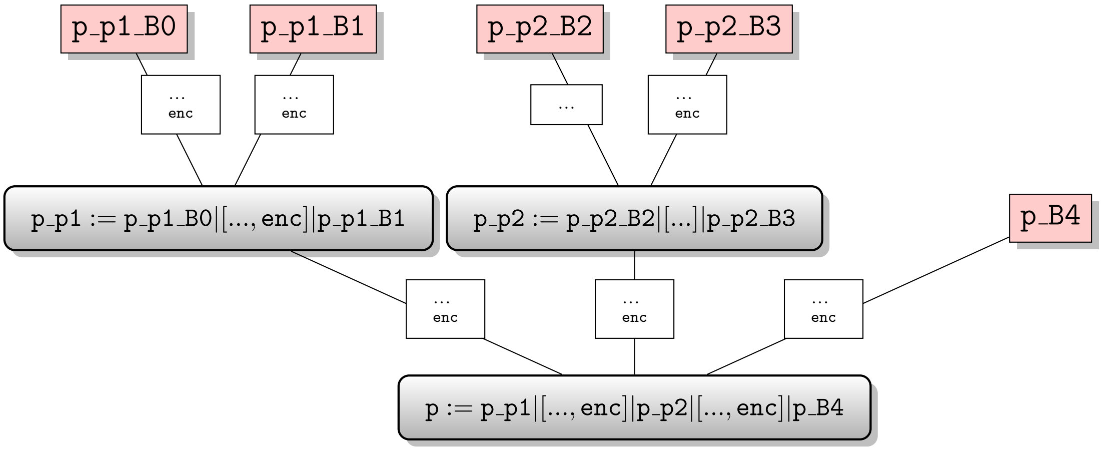
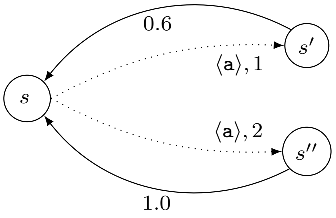

p_p1_B0
p_p1_B1
p_p2_B2
p_p2_B3
... enc
... enc
...
... enc
p_p1 := p_p1_B0|[..., enc]|p_p1_B1
p_p2 := p_p2_B2|[...]|p_p2_B3
p_B4
... enc
... enc
... enc
p := p_p1|[..., enc]|p_p2|[..., enc]|p_B4

Figure 3.24: Process composition regarding the encoded action prefixes

The generated SPA processes are then binarily composed regarding the given instance structure, as shown by Figure 3.24. All leaf nodes depict the processes translated from the Behavior instances. The labels of their action prefixes serve as unique encodings of generative/reactive product term combinations. When a process is involved in an interaction (i.e. it is addressed by a generative/reactive product term encoded by enc), the action label enc is offered to the environment. Whenever it is composed with another process which reveals the same encoding, the label is included in the synchronisation set of the composition node. Since process p corresponds to the Module instance that defines the guards statement from which all literals encoded with enc originate, enc is provided by each involved process within the composition substructure of p.

Since a product term splits into literals which may be negated or unnegated, each addressed Behavior instance will translate to an SPA process containing action prefixes which are labelled by the arising product term encoding. The composition operators among the


0.6
$\langle \mathtt{a}\rangle, 1$
s'
s
$\langle \mathtt{a}\rangle, 2$
s''
1.0

(a) Behavior definition of B0

```
p_p1_B0(state[2]) :=
  [state = s ]  (*enc, 1*); p_p1_B0(s') + (*enc, 2*); p_p1_B0(s'')
  [state = s' ]  ...
  [state = s'']  ...
```

(b) Translated sequential SPA process for B0

Figure 3.25: Example of a Behavior instance translation into a sequential SPA process

processes hereby serve as conjunction operators stating that a product term is satisfied in case that all processes can perform an action prefix of a specific encoding.

If a process with multiple action prefixes (of a single encoding) of a choice is composed with another process that also offers this label by multiple action prefixes, the composed process will comprise the product of these actions sets. Let an example be given, where the instance B0 arises from the Behavior definition depicted by Figure 3.25(a). Moreover, the instance B1 arises from a differently parametrised variant of this Behavior definition, such that $s \xrightarrow{\langle \mathtt{a}\rangle,3} s'$ and $s \xrightarrow{\langle \mathtt{a}\rangle,5} s''$. Therefore, each instance can reach s' or s'' by triggering $\langle \mathtt{a}\rangle$ via some product term (encoded by enc). When both translated sequential SPA processes (e.g. Figure 3.25(b)) are composed, the resulting state space, as depicted by Figure 3.26, finally reveals 4 transitions which are part of a choice.

When considering the LTS semantics, a composed state (s_i, s_j) can be derived which corresponds to the composed process of the left-hand-side of Figure 3.26:

$$\underbrace{(*\mathtt{enc},1*);\mathtt{p\_p1\_B0}(\mathtt{s}') + (*\mathtt{enc},2*);\mathtt{p\_p1\_B0}(\mathtt{s}'')}_{s_i} \quad |[\mathtt{enc}]| \quad \underbrace{(*\mathtt{enc},3*);\mathtt{p\_p1\_B1}(\mathtt{s}') + (*\mathtt{enc},5*);\mathtt{p\_p1\_B1}(\mathtt{s}'')}_{s_j}$$

The sets of jointly acting transitions are determined by the possible transitions of each addressed Behavior instance. For this reason, a number of transitions can be derived such that a one-to-one correspondence can be found with regard to the transitions obtained by the SPA semantics.

In order to generalise this, it is necessary to show that the state space implied by the semantic rules for the SPA translation of LARES by taking the CASPA LTS semantics into account is equivalent to the state space directly obtained by applying the LARES LTS rules. This can be done by pointing out the correspondence between these rules.

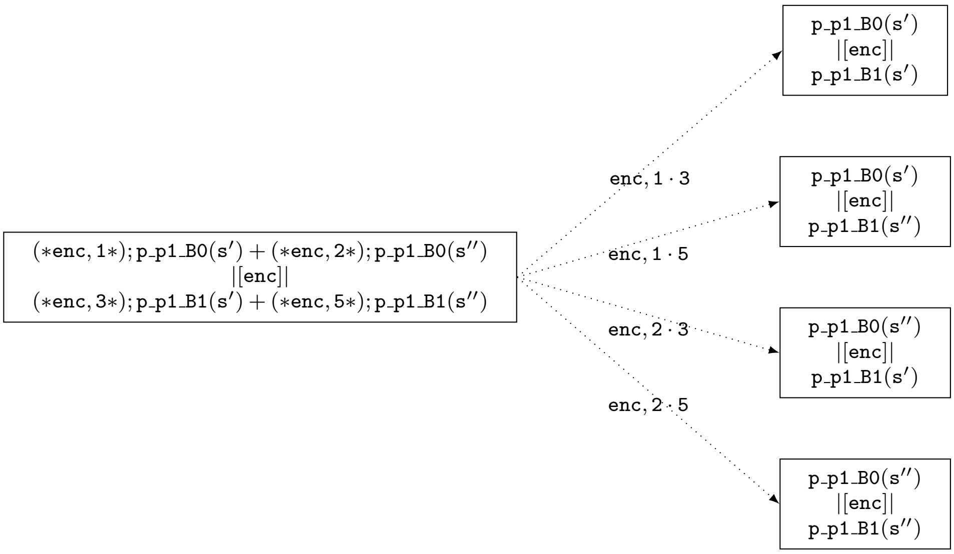

(∗enc, 1∗); p_p1_B0(s′) + (∗enc, 2∗); p_p1_B0(s″)
||[enc]||
(∗enc, 3∗); p_p1_B1(s′) + (∗enc, 5∗); p_p1_B1(s″)
enc, 1 · 3
enc, 1 · 5
enc, 2 · 3
enc, 2 · 5
p_p1_B0(s′) ||[enc]|| p_p1_B1(s′)
p_p1_B0(s′) ||[enc]|| p_p1_B1(s″)
p_p1_B0(s″) ||[enc]|| p_p1_B1(s′)
p_p1_B0(s″) ||[enc]|| p_p1_B1(s″)

Figure 3.26: Resulting state space due to SPA composition of B0 and B1

Formalising the Proof Sketch

The formal representation of the LARES SPA and LTS semantics in terms of SOS rules is given in the sections 3.4.3 and 3.5.4 respectively. If a state of an SPA process satisfies one of the given SPA transformation rules, at least one action prefix will be offered to which other processes can synchronise. According to this, let a product term combination (pt_g, pt_r) be given which originates from a guards statement. Following Section 3.5.4, the product term is encoded by enc and transformed into the sets L_g and L_r of augmented literals. Each augmented literal is then forwarded towards its addressed Behavior instance, as illustrated by Figure 3.23 where e.g. $\{(\mathtt{enc}, \mathtt{B0}.\langle\mathtt{a}\rangle), (\mathtt{enc}, \mathtt{B1}.\langle\mathtt{a}\rangle)\}$ represents the set of reactive literals which have reached p1.

Let B_i and B_j be two distinct Behavior instances (such that $i \neq j$). The set of augmented literals which reach B_i are denoted by $L_g|_i$ and $L_r|_i$, whereas the set of augmented literals for B_j are denoted by $L_g|_j$ and $L_r|_j$. When referring to the SOS rules which denote the SPA semantics of a LARES$_{\text{BASE}}$ model, the term *local satisfiability* is used with regard to the question whether a state s_i of a Behavior B_i satisfies $L_g|_i$ and $L_r|_i$. If the interaction flag f inside the local augmented literals (originating from a single local product term, e.g. for $(L_g|_i, L_r|_i)$), is $false$ (which is the trivial case), no real interaction with other components will take place, as enc will not be part of the synchronisation information of the Behavior's $\mathcal{PAT}$ representation. Otherwise, a real interaction which involves several Behaviors will take place. Accordingly, enc will be part of the synchronisation information of the Behavior's $\mathcal{PAT}$ representation. The remainder of this section only deals with real interactions (i.e. the non-trivial case, where the interaction flag is $true$).

The PACT approach unifies the synchronisation information along the composition structure (cf. Section 3.5.3). The actions of two processes to be synchronised are determined by the set of labels given by the intersection of the process synchronisation information (see Figure 3.24 for illustration). For this reason, it holds that a process s_x which reveals some specific behaviour (e.g. it is able to perform an action labelled by `enc`) will still behave like s_x with regard to the behaviour `enc` (which is still part of the synchronisation information) even though the process is composed with another process. Let s_y, s_z be further processes which reveal some desired action `enc`. Irrespective of the given composition structure, the associativity property holds when considering the behaviour encoded by `enc`:

$$(s_x|[enc]|s_y)|[enc]|s_z = s_x|[enc]|(s_y|[enc]|s_z) = s_x|[enc]|s_y|[enc]|s_z$$

Whenever a state s_i, which can also be seen as a process term, of a Behavior instance B_i satisfies the augmented literals of a specific encoding `enc`, an action prefix $\overset{enc}{\rightsquigarrow}_{d_i}; s_i'$ is constructed for that process. In case that this action prefix can be performed together with another process s_j, which also reveals an enabled action prefix of that encoding, the following rule applies and eventually derives a composed transition from this pair of enabled action prefixes:

$$\frac{\dfrac{\overbrace{s_i \models L_g|_i \wedge s_i \models L_r|_i}^{\text{rules (3.12) or (3.11)}}}{\overset{enc}{\rightsquigarrow}_{d_i}; s_i' \overset{enc,d_i}{\dashrightarrow} s_i'} \; s_i = \overset{enc}{\rightsquigarrow}_{d_i}; s_i' \qquad \dfrac{\overbrace{s_j \models L_g|_j \wedge s_j \models L_r|_j}^{\text{rules (3.12) or (3.11)}}}{\overset{enc}{\rightsquigarrow}_{d_j}; s_j' \overset{enc,d_j}{\dashrightarrow} s_j'} \; s_j = \overset{enc}{\rightsquigarrow}_{d_j}; s_j'}{\underbrace{\overset{enc}{\rightsquigarrow}_{d_i}; s_i' \;|[enc]|\; \overset{enc}{\rightsquigarrow}_{d_j}; s_j'}_{s_i \;|[enc]|\; s_j} \overset{enc,\phi(d_i,d_j)}{\dashrightarrow} s_i' \;|[enc]|\; s_j'} \quad j \neq i \tag{3.13}$$

Accordingly, the composed process $s_{\{i,j\}} = s_i \;|[enc]|\; s_j$ satisfies the augmented literals of $L_g|_{\{i,j\}}$ and $L_r|_{\{i,j\}}$ regarding `enc`. For $s'_{\{i,j\}} = s_i' \;|[enc]|\; s_j'$ the above rule can be abstracted to

$$\frac{s_{\{i,j\}} \models L_g|_{\{i,j\}} \quad \wedge \quad s_{\{i,j\}} \models L_r|_{\{i,j\}}}{s_{\{i,j\}} \overset{enc,\phi(d_i,d_j)}{\dashrightarrow} s'_{\{i,j\}}} \quad s_i = \overset{enc}{\rightsquigarrow}_{d_i}; s_i' \;\wedge\; s_j = \overset{enc}{\rightsquigarrow}_{d_j}; s_j'$$

It is important to note that the multiplicity of transitions of a choice in the composed state space depends on the number of occurrences of addressed guarded transitions in a choice of a single Behavior instance. Since an application of the rule (3.11) treats a guarded transition (addressed by an unnegated reactive literal) as an action prefix, the occurrence of multiple enabled guarded transitions is hence captured by a choice of multiple action-prefixed processes.

According to that, an example will be given. Let the SOS rule (3.11) apply to process s_i which reveals two guarded transitions (labelled l_i' and l_i'' that are addressed by unnegated reactive minterm literals). Within process s_j only one guarded transition is addressed, which leads to a single enabled action. As a result, two transitions (similar to the ones depicted in Figure 3.26) will arise as a choice in the resulting state space:

$$\dfrac{\overbrace{\dfrac{s_i \models L_g|_i \wedge s_i \models L_r|_i \quad s_i \begin{array}{l} \xrightarrow{\langle \mathtt{l}'_\mathtt{i} \rangle, d'_i} s'_i \\ \xrightarrow[\langle \mathtt{l}''_\mathtt{i} \rangle, d''_i]{} s''_i \end{array}}{\overset{enc}{\rightsquigarrow}_{d'_i}; s'_i + \overset{enc}{\rightsquigarrow}_{d''_i}; s''_i \begin{array}{l} \xrightarrow{enc, d'_i} s'_i \\ \xrightarrow[enc, d''_i]{} s''_i \end{array}}}^{\text{rule 3.11}} \qquad \overbrace{\dfrac{s_j \models L_g|_j \wedge s_j \models L_r|_j \quad s_j \xrightarrow{\langle \mathtt{l}_\mathtt{j} \rangle, d_j} s'_j}{\overset{enc}{\rightsquigarrow}_{d_j}; s'_j \xrightarrow{enc, d_j} s'_j}}^{\text{rule 3.11}}}{\left(\overset{enc}{\rightsquigarrow}_{d'_i}; s'_i + \overset{enc}{\rightsquigarrow}_{d''_i}; s''_i\right) \ |[enc]| \ \overset{enc}{\rightsquigarrow}_{d_j}; s'_j \begin{array}{l} \xrightarrow{enc, \phi(d'_i, d_j)} s'_i|[enc]|s'_j \\ \xrightarrow[enc, \phi(d''_i, d_j)]{} s''_i|[enc]|s'_j \end{array}} \tag{3.14}$$

By each composition, where multiple action prefixes are in a choice, the number of composed transitions increases following the cross-product of the arising combinations of potentially cooperating actions.

By an inductive application of rule (3.13), a product term combination can be successively satisfied regarding the composed state space. This will be the case, if the associated synchronised action can be performed at the current state, composed by the current states of the addressed Behavior instances. Let A denote all addressed processes of Behavior instances. The inductive application of rule (3.13) leads to a rule that considers only a single superior process s as an abstraction from the composition structure encompassing all addressed instances s_a (where $a \in A$), such that $s = \{s_a \mid a \in A\}$.

The process s hereby satisfies a complete product term combination by being able to perform the synchronised action $s \xrightarrow{enc, d_A} s'$. The composed distribution is determined by a function $\phi : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ defined as $\phi : (d_1, d_2) \mapsto d_1.value \cdot d_2.value$ which multiplies the values of two equally typed distributions d_1, d_2, such that either two rates or two weights are multiplied. As both rates and weights are real valued numbers, the associativity property holds for ϕ:

$$\phi(\phi(d_i, d_j), d_k) = \phi(d_i, \phi(d_j, d_k)) = \phi(d_i, d_j, d_k)$$

The composed distribution d_A can consequently be calculated by $d_A = \phi(\{d_a\}_{a\in A})$ and the generalised rule can be derived as follows:

$$\frac{\forall_{a\in A}\ \dfrac{\overbrace{s_a \vDash L_g|_a \wedge s_a \vDash L_r|_a}^{\text{rules (3.12) or (3.11)}}}{\overset{enc}{\rightsquigarrow}_{d_a};s'_a \xrightarrow{enc,d_a} s'_a}\ s_a = \overset{enc}{\rightsquigarrow}_{d_a};s'_a}{s \xrightarrow{enc,d_A} s'}\quad \begin{matrix} s{=}\{s_a \mid a{\in}A\} \wedge s'{=}\{s'_a \mid a{\in}A\} \wedge \\ d_A{=}\phi(\{d_a\}_{a\in A}) \end{matrix} \tag{3.15}$$

Rule 3.15 can be simplified such that the composed process s satisfies all augmented literals $L_g = L_g|_{\{a\in A\}}$ and $L_r = L_r|_{\{a\in A\}}$ of a product term combination:

$$\frac{s \vDash L_g \wedge s \vDash L_r}{s \xrightarrow{enc,d_A} s'}\quad \begin{matrix} \forall_{a\in A}\ s_a{=}\overset{enc}{\rightsquigarrow}_{d_a};s'_a \wedge \\ s{=}\{s_a \mid a{\in}A\} \wedge s'{=}\{s'_a \mid a{\in}A\} \wedge d_A{=}\phi(\{d_a\}_{a\in A}) \end{matrix} \tag{3.16}$$

When abstracting from the side-conditions, the following holds

$$\text{rule (3.16)} \triangleq \frac{s \vDash L_g \wedge s \vDash L_r}{s \xrightarrow{enc,d_A} s'} \xleftrightarrow{\substack{L_g \triangleq pt_g \\ L_r \triangleq pt_r}} \frac{s \vDash pt_g \wedge s \vDash pt_r}{s \xrightarrow{enc,d_A} s'} \triangleq \text{rules (3.6) and (3.7)}$$

When taking the rules' side-conditions into account, the number of composed transitions due to the applications of the SPA rules derives from the cross-product of the number of addressed guarded transitions (by unnegated reactive literals), as illustrated by the substitution of rules in (3.14). Equivalently, the LTS semantics defines a number of composed transitions $T_c \in T_{combinations}$ (which are in a choice), derived from the cross-product of the addressed transitions within each sequential process of a Behavior instance (cf. Eq. (3.4)). For this reason, the number of jointly acting transitions is, as defined used by the LTS rules, equivalent to the number of composed transitions originating from the combinatorics of all possible SPA rule applications. Furthermore, the calculation of composed distribution (by the application of the SPA rules) matches exactly the calculation of rates or weights for the LTS rule as given by Eq. (3.5).

The reachability analysis eventually yields a bijective relation between all composed processes (originating from the SPA semantics) and the composed states (originating from the LTS semantics). Following the above arguments, also among the transitions (which are constructed due to the composition rules of the SPA semantics or the LTS rules) a one-to-one correspondence can be found (such that pairs of transitions can be built which have equivalent attributes, regarding their source/target state, distribution type and label). Accordingly, the application of the SPA rules and the application of the LTS rules yield an isomorphic transition system which is behaviourally equivalent and hence ultimately a special case of bisimulation. □

Chapter 4

Structure and Implementation of the LARES Framework

The current LARES framework consists of a number of implementations which constitute the integrated development environment (IDE) for LARES as described in Section 4.1. It also incorporates a library developed to provide the language definitions and algorithms to deal with models of fault-tolerant systems (denoted as fault-tolerant system environment `ftse`, further detailed in Section 4.2). Beside the library components related specifically to LARES, parts which can be used independently of LARES are available such as different meta-models (which serve as input languages to certain solvers), extensible sublanguages (which can be embedded into arbitrary language definitions) or generic algorithms (e.g. for calculation of bisimulation equivalence between two models).

Figure 4.1 shows how the Eclipse-based IDE components and the library parts are linked. Furthermore, references to the sections which define the theoretical aspects corresponding to the items inside the figure are provided. Note that the transformation from LARES$_{\text{FLAT}}$ into SPN as well as the transformation into the MDP formalism is not part of this work (regarding the MDP transformation cf. [69, 70] instead).

4.1 Components of the Eclipse-based LARES IDE

The LARES IDE comprises several individual Eclipse plugins [66, 67], which become apparent in the screenshot shown in Figure 4.2, i.e.

- the implementation of a textual Editor Plugin with syntax highlighting, code completion and validation features for syntax and context conditions (for serving as a comfortable modelling environment which is detailed in Section 4.1.1) and

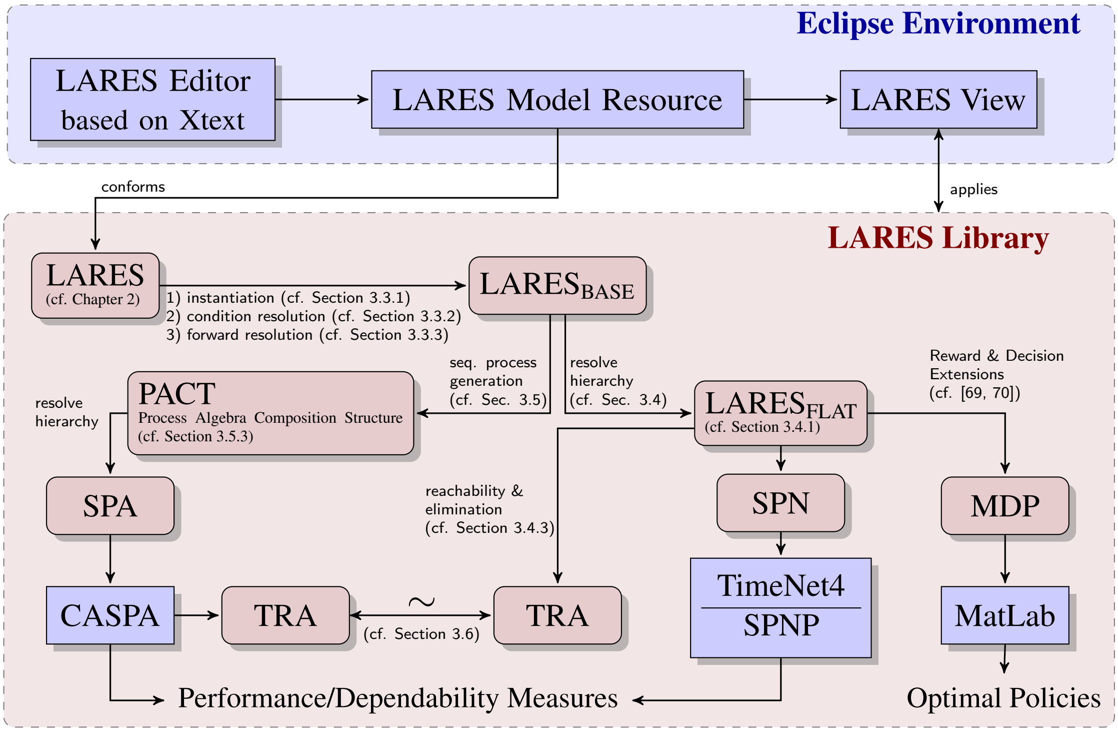

Eclipse Environment
LARES Editor based on Xtext
LARES Model Resource
LARES View
conforms
applies
LARES Library
LARES (cf. Chapter 2)
1) instantiation (cf. Section 3.3.1)
2) condition resolution (cf. Section 3.3.2)
3) forward resolution (cf. Section 3.3.3)
LARES_BASE
seq. process generation (cf. Sec. 3.5)
resolve hierarchy (cf. Sec. 3.4)
Reward & Decision Extensions (cf. [69, 70])
resolve hierarchy
PACT
Process Algebra Composition Structure (cf. Section 3.5.3)
LARES_FLAT (cf. Section 3.4.1)
SPA
reachability & elimination (cf. Section 3.4.3)
SPN
MDP
CASPA
TRA
~
(cf. Section 3.6)
TRA
TimeNet4
SPNP
MatLab
Performance/Dependability Measures
Optimal Policies

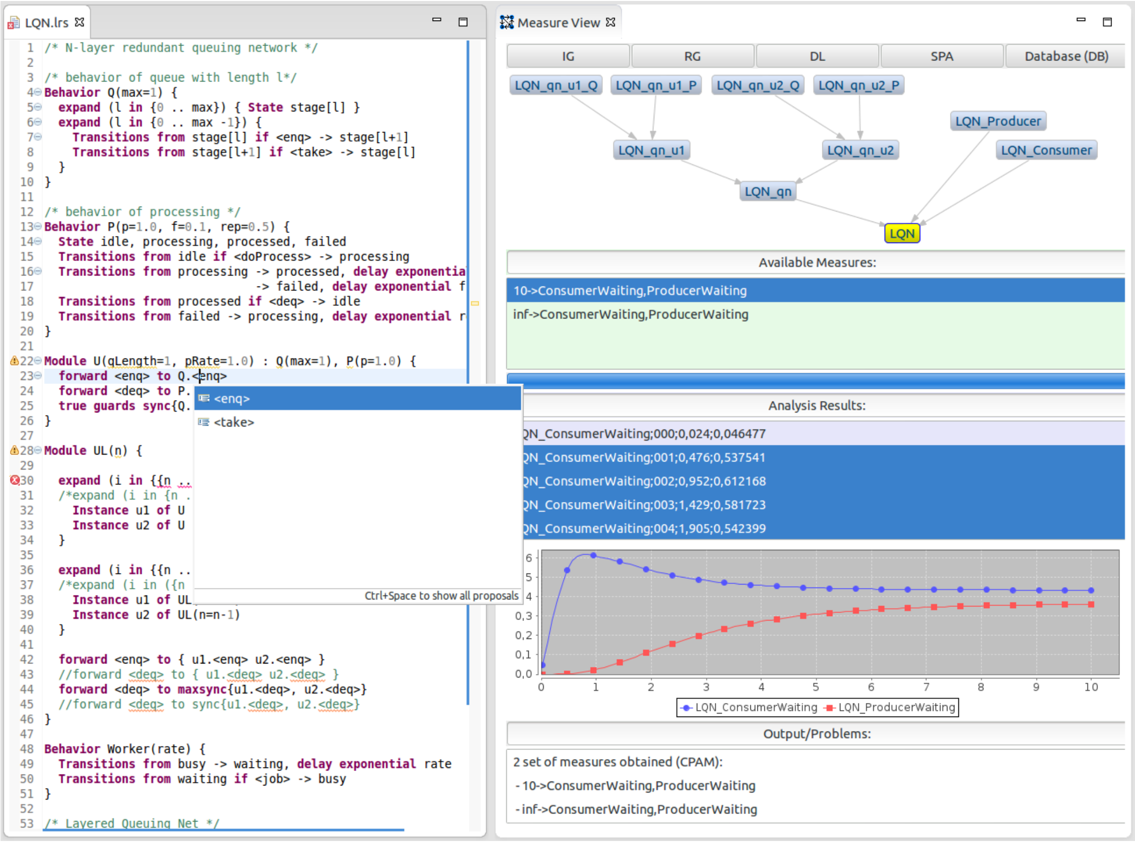
Figure 4.1: Overview of the LARES IDE implementation

- the LARES View Plugin which has been developed to carry out the analysis of measures, to define experiments and to manage them (as detailed in Section 4.1.3).

For example, in the Editor Plugin (cf. left-hand-side of Figure 4.2) a context menu can be opened which offers the visible guard labels in order to ease specifying a forward statement. Furthermore, the LARES View Plugin (on the right-hand-side) is responsible for visualising the instance tree of a model, providing a choice among the specified measures in order to select and to perform the desired type of analysis, or showing the results directly by a 2D plot. In addition, Section 4.1.2 describes a graphical editor which is currently under development and will serve in the near future as a complement to the textual editor. There are ideas to integrate the graphical editor into the textual one in order to enable consistency preserving hybrid editing.

4.1.1 Textual Editor Plugin

As the grammar in Chapter 2 is defined in terms of syntactical rules following the Xtext notation, an editor plugin can be directly derived, as described in [58]. Each rule, written in the Xtext formalism, consists of a name and may state several terminal rules (such as identifiers or numbers) or non-terminal rules (referring to other rules).



```
/* N-layer redundant queuing network */

/* behavior of queue with length l*/
Behavior Q(max=1) {
  expand (l in {0 .. max}) { State stage[l] }
  expand (l in {0 .. max -1}) {
    Transitions from stage[l] if <enq> -> stage[l+1]
    Transitions from stage[l+1] if <take> -> stage[l]
  }
}

/* behavior of processing */
Behavior P(p=1.0, f=0.1, rep=0.5) {
  State idle, processing, processed, failed
  Transitions from idle if <doProcess> -> processing
  Transitions from processing -> processed, delay exponentia
                              -> failed, delay exponential f
  Transitions from processed if <deq> -> idle
  Transitions from failed -> processing, delay exponential r
}

Module U(qLength=1, pRate=1.0) : Q(max=1), P(p=1.0) {
  forward <enq> to Q.<enq>
  forward <deq> to P.
  true guards sync{Q.
}

Module UL(n) {

  expand (i in {{n ..
  /*expand (i in {n .
    Instance u1 of U
    Instance u2 of U
  }

  expand (i in {{n ..
  /*expand (i in ({n
    Instance u1 of UL
    Instance u2 of UL(n=n-1)
  }

  forward <enq> to { u1.<enq> u2.<enq> }
  //forward <deq> to { u1.<deq> u2.<deq> }
  forward <deq> to maxsync{u1.<deq>, u2.<deq>}
  //forward <deq> to sync{u1.<deq>, u2.<deq>}
}

Behavior Worker(rate) {
  Transitions from busy -> waiting, delay exponential rate
  Transitions from waiting if <job> -> busy
}

/* Layered Queuing Net */
```

10->ConsumerWaiting,ProducerWaiting
inf->ConsumerWaiting,ProducerWaiting

```
QN_ConsumerWaiting;000;0,024;0,046477
QN_ConsumerWaiting;001;0,476;0,537541
QN_ConsumerWaiting;002;0,952;0,612168
QN_ConsumerWaiting;003;1,429;0,581723
QN_ConsumerWaiting;004;1,905;0,542399
```

2 set of measures obtained (CPAM):
- 10->ConsumerWaiting,ProducerWaiting
- inf->ConsumerWaiting,ProducerWaiting

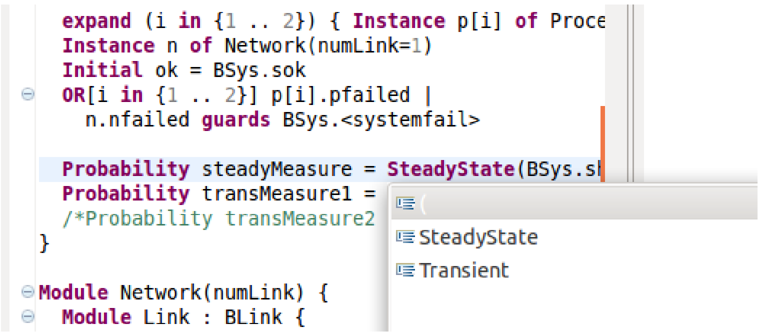
Figure 4.2: Screenshot of the LARES IDE showing the LARES Editor and the View Plugin

For illustration purposes, two grammar rules which partially define the syntax of a Transitions statement are consulted:

```
TransitionsStmt:
  "Transitions" "from" s = ID (transitions += Transition)+;
Transition:
  ("if" '<' g = ID '>')?  '->' t = ID (',' d = Distribution)?;
```

The application of the rule `TransitionsStmt` requires an occurrence of “`Transitions from`” in order to parse an identifier corresponding to the name of a source state which is stored in the attribute s of the resulting model object for a Transitions statement. Several transitions are parsed by the referred rule `Transition` and stored in the attribute *transitions* of the Transitions statement model object. The `Transition` rule allows a guard label to be defined by first parsing an “`if`” followed by an identifier that is indicated in brackets and stored within the guard label attribute denoted as g. An optional element is specified by ‘?’. The string “`->`” indicates that a target state follows which is then stored in attribute ‘t’. Furthermore, a distribution can optionally be specified, which is not further detailed. The rules shown above define the concrete syntax, but they also define the abstract model in terms of an Ecore meta-model [57].

Alternatively, if one desires a looser relationship between the concrete syntax and its abstract representation, it is also possible to first define a meta-model whose elements are instantiated by the application of Xtext rules and thus result in an overall model conforming to the given meta-model.



```
  expand (i in {1 .. 2}) { Instance p[i] of Proce
  Instance n of Network(numLink=1)
  Initial ok = BSys.sok
  OR[i in {1 .. 2}] p[i].pfailed |
    n.nfailed guards BSys.<systemfail>

  Probability steadyMeasure = SteadyState(BSys.s
  Probability transMeasure1 =
  /*Probability transMeasure2
}

Module Network(numLink) {
  Module Link : BLink {
```

Figure 4.3: Syntax highlighting and code completion of the LARES Editor

A LARES Editor component which is generated by using only rules of the previously illustrated type provides a number of features by default such as code completion, syntax highlighting (cf. Figure 4.3) and indication of syntax error positions. Moreover, an outline view depicts the structure of the constructed internal model representation. In addition to extending the standard features such that they behave in a smarter way, the LARES Editor Plugin has been enriched by sophisticated modelling template support and validation features in order to make it even more user-friendly [66, 71]. Therefore, the *scoping functionality* to restrict cross-referable objects has been refined such that it now considers the hierarchic nature of LARES and implements its visibility constraints. For this purpose, the grammar rules also have been adapted by using cross-references. The standard *hyperlinking feature* then allows navigating through the textual model along the referred objects. As LARES models may also be parametrisable, finding a referred object is a non-trivial task. Therefore, the standard *linking feature* is specialised such that the indices of the references are pattern matched against those of the referred object to provide either the correct link or a provisional link to the supposed matched object. If no pattern can be applied, the model will be invalid. Apart from the *syntax validator*, which checks whether a textual model conforms to the grammar specification of LARES, additional semantics-related validation capabilities can be performed, e.g. as already mentioned by the provisioning of cross-references. Moreover, other semantic issues can be detected with regard to uniqueness of identifiers in scope (for all named statements or definitions), the domain of variables in the context of an expression (e.g. a weight or a rate have to be positive), or cyclic dependencies in variable definitions (e.g. of the iterator variables in an expand statement). All of these features are performed on-the-fly when modelling and are therefore required to be fast. In consequence, the linking feature does not resolve the whole system parametrisation by evaluating index variables, accepting the downside of not detecting all invalid references. However, this can be done offline. Therefore, the model is transformed (using the EPL pattern matching language in Eclipse Epsilon [55]) in order to obtain a parameter-resolved representation, where all variables are evaluated and thus allow a proper validation thereon (i.e. cross-reference checks, variable domains, etc.). It needs to be mentioned that dead code segments in a model (i.e. abstract definitions that have not been addressed by any instantiation) cannot be considered properly for validation as variables might remain undefined. Furthermore, the *code completion* feature has been refined from the standard implementation in order to context-specifically reduce the number of possible suggestions, *predefined templates* further improve the time-efficiency and convenience of the modelling process while a *formatter* performs *pretty printing* of the specified code following the common conventions for programming.

4.1.2 Graphical Editor Plugin

In [61] an attempt was made to develop a graphical editor plugin for the LARES language using the Eclipse Framework GMF which became part of the Graphical Modeling Project [73]. In contrast to the expertise gained by the development of a GMF-based editor for the process algebra for CASPA [10, 11], it turned out that the model-driven approach of GMF requires huge effort when used for hierarchical languages as it needs a lot of on-the-fly information that can only be provided by hand-crafted implementations. This, in consequence, takes the idea of a model driven approach ad absurdum. Furthermore, the GMF framework lacks support to conveniently adapt to changes and to remain extensible in a simple, straightforward manner. Due to limitations of resources in time, the work started in [61] could not be completed.

Even so, the desire to implement a graphical editor for the LARES language was undaunted, despite the fact that the mentioned model-based approach could not be used stringently. As an alternative approach, the Graphiti Framework [56] developed by SAP, which is also hosted as a project of the Eclipse Foundation, has shown to serve as a good starting point for a graphical editor by providing a plain Java interface and support for the Eclipse Modelling Framework (EMF) [57]. In addition, when opening a LARES model that has been specified textually beforehand, the ability of Graphiti to use existing layout algorithms allows a satisfiable initial arrangement to be generated. The current state of development is described in [66]. The graphical editor implementation hereby applies the Graphiti classes which have been extended in order to realise the graphical notation and tooling. From the LARES standard meta-model, the corresponding Java classes have been generated to be used for model instantiation. Standard features such as *delete*, *resize* and wizards for constructing diagrams including the tooling could directly be applied, reducing the amount of development time. The implementation to consistently manage resources within a consistent central resource set, to store hierarchic LARES models (where the layout information is separated from the content of a model) and to *drill-down* (construct/open new diagrams from the inside of another diagram via double-click) is challenging. All this is still ongoing work.

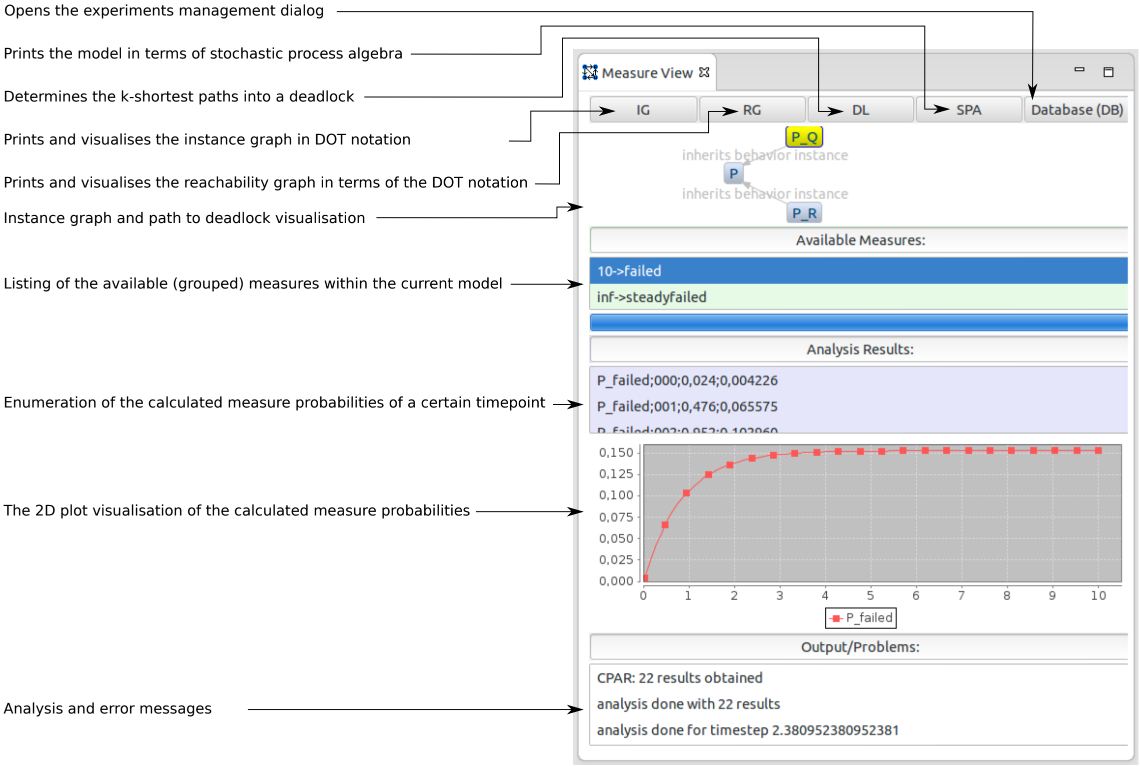
4.1.3 LARES View Plugin

While the objective of the editor plugins described in Section 4.1.1 and 4.1.2 is to specify a model which conforms to the LARES syntax and allows the semantics to be partially validated, a further plugin is responsible to support the experimentations by carrying out the analysis process, organising parametrisations and results. The plugin was developed as a View Plugin component for the Eclipse environment. It defines and implements a simple and intuitive user interface which allows

- performing the analysis and managing the experiments as well as
- visualising the experimental results (by 2D plots, graphs and others)

When a LARES model becomes active within the current editor page or in case a modification is applied, the LARES View Plugin will react on these events to process the editor's content. If the content turns out to be a valid LARES model, the instance tree will be constructed following the transformation described in Section 3.3.1. Instantaneously a DOT graph [9] is generated from the instance tree and visualised on a SWT Composite [133] inside the plugin's GUI (by applying the ZEST library [72] which incorporates a graph layout engine).

If the LARES model contains Probability statements, the LARES View Plugin will extract and list these statements in the corresponding SWT Composite of the GUI. All transient measures are grouped (corresponding to the specified points in time) in order to calculate each group of measures within a single run by executing the CASPA analysis backend. All steady state measures are also pooled within one group. When selecting a group of transient measures, a dialogue box asks the user to specify a number of intermediate time points (between 0 and the point in time corresponding to the group of measures) to be calculated. These are used in order to determine the temporal course of a group of measures at a certain level of granularity. The whole transformation workflow is then executed to generate the SPA specification which is subsequently completed by the selected and translated measures and redirected into the CASPA solver. The calculated results are collected in a list of items on the plugin GUI. A copy&paste feature has been implemented therefore in order to facilitate the transfer of the results to other external tools.



Opens the experiments management dialog

Prints the model in terms of stochastic process algebra

Determines the k-shortest paths into a deadlock

Prints and visualises the instance graph in DOT notation

Prints and visualises the reachability graph in terms of the DOT notation

Instance graph and path to deadlock visualisation

Listing of the available (grouped) measures within the current model

Enumeration of the calculated measure probabilities of a certain timepoint

The 2D plot visualisation of the calculated measure probabilities

Analysis and error messages

Available Measures:

10->failed
inf->steadyfailed

Analysis Results:

P_failed;000;0,024;0,004226
P_failed;001;0,476;0,065575
P_failed;002;0,952;0,102960

Output/Problems:

CPAR: 22 results obtained
analysis done with 22 results
analysis done for timestep 2.380952380952381

Figure 4.4: Explanation of the LARES View Plugin GUI

Furthermore, a 2D plot of the measures is drawn directly using the JFree chart library [89]. If a positive number of intermediate time points has been specified, a parallel execution of the CASPA instances will be performed according to the number of logical cores of the CPU. In addition, the DOT representations for instance graphs and transition systems can be visualised and serialised directly from the GUI. Similarly, the 2D plot of the temporal course of the measures can be serialised in terms of the SVG standard [137] and the generated SPA specification can be stored as a simple text file. It is planned to integrate a database which allows managing the experiments performed on the models and their different versions and parametrisations. Figure 4.4 explains the placement and responsibility of each GUI element. Different dialogues which can be opened to perform an analysis or to store the 2D plot are not detailed as they are self-explaining due to their simplicity.

Internally, the LARES View Plugin is a mixed Scala/Java project which applies the Akka actor concept [135] in order to ensure smooth usability of the plugin and the whole Eclipse environment under the load of an analysis. A further benefit of using actors is that they communicate by message passing. This leads to a decoupling of the internal components. Other views can be easily implemented to add further functionality.

4.2 The Scala-based Library Implementation

Scala [110] has been employed for the whole transformation implementation. Extensive use of some of its functional and OO concepts has been made. An abstract syntax tree definition which corresponds to the formal definitions made in 2.2 has been built using algebraic data types (in terms of Scala case classes). Furthermore, a concrete syntax has been implemented using Scala parser combinators. The abstract syntax tree is created, i.e. a model is loaded, by applying the root parser to a LARES specification. Next, the transformation has to be performed. Classically the visitor pattern is applied to ensure separation between the abstract syntax tree implementation and the transformation code. As the tree consists of algebraic data types, Scala pattern matching is used for decomposition instead of traversing the tree applying the visitor pattern in order to retain this separation. All described transformations have been implemented based on the case class structures which represent the source or the target formalism. Eventually, a simple model-to-text transformation is carried out on the target model. This transformation yields an SPA model which can be directly forwarded to the CASPA solver.

The relevant aspects of the LARES library subdivides into the following sub-projects:

`ftse-base` LARES-independent embeddable extensible language representations, parsers, serialisers and algorithms (e.g. for arithmetic, logical and set theoretic expressions, the associated evaluation algorithms, bisimulation algorithms, and abstract representations for CASPA SPA, TimeNet Petri nets, LTS, etc)

`ftse-lares` LARES abstract representation, LARES$_{\mathrm{FLAT}}$ representation and the embedded language extensions, parsers, serialisers, transformation algorithms from LARES to LARES$_{\mathrm{BASE}}$, from LARES$_{\mathrm{BASE}}$ to CASPA SPA, or from LARES$_{\mathrm{FLAT}}$ to TimeNet or, by performing a reachability algorithm, to an LTS

`ftse-lares-extensions` LARES related extensions such as rewards or non-deterministic decisions, extended parsers, serialisers and transformations

`ftse-lares-models` A set of LARES models which are used to unit-test parsers and transformations, or which serve as case-studies

`ftse-lares-check` Unit-tests for parsers, transformations etc.

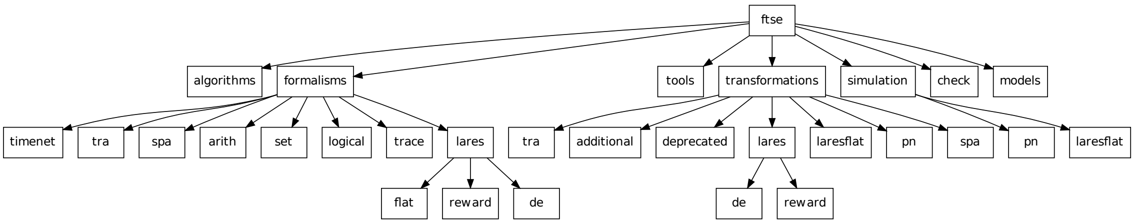


Figure 4.5: Package structure of the LARES library

All sub-projects incorporate and contribute to a given package structure as shown in Figure 4.5. The main `ftse` package is subdivided into the packages `algorithms` (general algorithms, e.g. for topological sorting or cross-product generation of elements), `formalisms` (to specify languages for common expressions, e.g. arithmetic expressions), `tools` (providing mainly classes and methods to wrap the interaction with the external command line tools), `transformations` (to bridge from an input model to a target model), `simulation` (contains the reachability algorithm), `check` (provides the unit-tests to check whether parsers work correctly or transformations produce meaningful results) and `models` (containing the models specified in terms of LARES or CASPA SPA).

When languages and their associated processing algorithms are intended to be extensible, it is crucial to avoid monolithic implementations. According to that, a general programming pattern in the Scala language, the stackable trait pattern, is introduced which is extensively used to achieve the required decoupling among the standard language and its extensions. Starting with Section 4.2.1, aspects of the library are detailed which define an abstract interface of a transformation and ease the chaining of different transformation steps (in order to compose the transformation workflows as described in Section 4.2.2).

Scala Stackable Trait Pattern

In order to allow building decoupled extensions of a basic transformation the so-called *Scala Stackable Trait Pattern* is used (cf. Section 12.5 in [110]). In Scala the concept of a *trait* is introduced to define object types which allow declaring method signatures and, in contrast to Java interfaces, implementing them.

For the sake of building an extensible transformation, a base object type is initially defined by means of the concept *trait* which has two type parameters `S` and `T` representing the source and the target model type respectively, and an abstract interface method `transform` which takes a source model `m` of type `S` into a model of type `T`:

```
trait AbstrTransformer[S,T] { def transform(m:S) : T }
```

In order to illustrate the use of the stackable trait pattern by example, a `transform` method implementation for LARES to LARES$_{\text{BASE}}$ is provided within a class definition. This definition serves as the core which inherits from the above basic trait `AbstrTransformer`:

```
class TrafoBase extends AbstrTransformer[LARES,LARESbase] {
  def transform(m:LARES) : LARESbase = ... // root method implementation
}
```

The above core class can subsequently be extended by *stackables*. Such extensions can be made by defining additional *trait*s which inherit from the above type object `AbstrTransformer`. The extended `transform` method is implemented such that the `super.transform` method of the superior trait or class definition is finally called in order to continue with the basic transformation or other extended transformations:

```
trait TrafoExt1 extends AbstrTransformer[LARES,LARESbase] {
  abstract override def transform(m : LARES) : LARESbase = {
    ...; super.transform(...)
  }
}
trait TrafoExt2 extends AbstrTransformer[LARES,LARESbase] {
  abstract override def transform(m : LARES) : LARESbase = {
    ...; super.transform(...)
  }
}
```

As shown subsequently, a (singleton) object `L2LbaseStd` representing the standard transformation can be instantiated from the core class `TrafoBase`, whereas all kinds of extensions can be *stacked* to e.g. construct an extended transformation object `L2LbaseExt` (which incorporates both extensions):

```
object L2LbaseStd extends TrafoBase
object L2LbaseExt extends TrafoBase with TrafoExt1 with TrafoExt2
```

Both objects (inside the preceding code-snipped) realise different transformations when performing the `transform` method. In general, the order of how traits are stacked and how the compiler performs linearisation (i.e. the rightmost trait is performed first) has an effect on the calculated result. However, as long as each implementation of the `transform` method (i.e. the root method or each extension method) does only process a distinct set of language elements, the transformation is independent of the order of stacking. Since the standard LARES language is intended to be extended by introducing new statements within the Module and Behavior definition bodies, the concept of processing distinct language elements by each stackable trait is applied to define the decoupled extensions to a number of model transformations. It is also used for embedded languages such as arithmetic expressions, where also transformations such as evaluations can be extended.

The following sections show how languages are extended, how they can be embedded into other languages, and how the corresponding algorithms can be extended (in a way that they remain decoupled using the stackable trait pattern) and composed easily.

4.2.1 LARES-independent Formalisms & Algorithms

As mentioned, the `ftse-base` sub-project implements a number of basic formalisms (such as logical expressions, set theoretic expressions and arithmetic expressions). The technical principles used for the implementation are the same for all of these formalisms. The arithmetic expression implementation has been chosen as a representative for all of them. Firstly, it is explained how to define its abstract representation and how this representation can be extended without losing type information in order to distinguish several kinds of arithmetic expressions. Secondly, it is shown how to define algorithms which process these structures and their extensions without losing compositionality of the algorithms.

Transformation Interface

Similarly to the example shown to illustrate the use of the stackable trait pattern, each transformation definition has to inherit the trait `AbstrTransformer[S,T]` which can be found in package `ftse.transformations` of sub-project `ftse-base`. The type parameter `S` denotes an abstract source type of a model, while `T` denotes a target type. Also here, an abstract function `transform(m:S):T` has to be specifically implemented.

Abstract Representation

In order to implement the abstract representation of arithmetic expressions, a trait with type parameter `T` is defined by `trait AE[T]`. Its type parameter is used to denote a specific type of arithmetic expression comprising a specialised structure. In its basic form an abstract representation of an arithmetic expression is defined as follows:

```
abstract class AE_Atom[T] extends AE[T]
abstract class AE_UnaryOp[T](t : AE[T]) extends AE[T]
abstract class AE_BinaryOp[T](l : AE[T], r : AE[T]) extends AE[T]
```

It consists of an abstract atomic element `AE_Atom[T]` and two basic kinds of abstract operators, i.e. unary and binary operators. An example of concrete representatives of the basic operators are defined as follows:

```
case class AE_Plus[T](l:AE[T], r:AE[T]) extends AE_BinaryOp[T](l,r)
case class AE_Min[T](l:AE[T], r:AE[T]) extends AE_BinaryOp[T](l,r)
case class AE_Mult[T](l:AE[T], r:AE[T]) extends AE_BinaryOp[T](l,r)
case class AE_Neg[T](t:AE[T]) extends AE_UnaryOp[T](t)
```

In order to give an example of how to define extensions based on the basic representation of `AE`[T], the traits `AE_ExtOp`[T] and `AE_ExtVal`[T] can be derived to denote the type of the extended expression (note that this is a constructed example):

```
trait AE_ExtOp[T] extends AE[T]
trait AE_ExtVal[T] extends AE[T]
```

Additional operators or atomic elements (e.g. `AE_Div`[T] or `AE_Value`[T]) may be specified for the extensions:

```
case class AE_Div[T](l:AE[T],r:AE[T]) extends
  AE_BinaryOp[T](l,r) with AE_ExtOp[T]
case class AE_Value[T](i:Int) extends AE_Atom[T] with AE_ExtVal[T]
```

Arbitrary arithmetic expressions with different capabilities can be composed following the above procedure. A specific parameter type for `T` can be defined for distinction, e.g. `trait AEext extends AE_ExtOp[AEext] with AE_ExtVal[AEext]`. Arithmetic expressions of different types (comprising different capabilities such as atomic elements or operators) can thus be constructed and distinguished in a type-safe way by the given parameter type, i.e. `AEext` for this example:

```
AE_Plus[AEext](AE_Value[AEext](1), AE_Value[AEext](2))
```

Decoupled Algorithms

Since a specification, and in this case the arithmetic expression as a representative, is required to be transformed or analysed, a traversal method needs to be implemented. An extension of such a language constitutes a new type of language (such as `AE`[`AEext`]). It is hence also desirable that the algorithms are decoupled and extensible. Again, the stackable trait pattern is used. Let a basic traversal class `AE_Transform` be defined with type parameters `T` which parametrises the type of arithmetic expressions and `R` for the type of the result. All elements inside an expression are traversed using the recursive `traverse` method implementation which applies pattern matching on each element in order to match to a specific language type (i.e. a *case class*) and to be hence able to apply the `modify` method and to perform the recursive descent. The `modify` method of an arithmetic expression for the basic language is herewith per default defined as the identity function, as no atomic elements (which are firstly introduced as specific extensions) are considered:

```
class AE_Transform[T,R] {
  def modify(ae : AE[T]) : R = ae
  def traverse(ae : AE[T]) : R = ae match {
    case AE_Plus(l,r) => modify(AE_Plus(traverse(l), traverse(r)))
    ...
    case sthelse => modify(sthelse)
  }
}
```

A stackable trait which enables the basic traversal algorithm in order to evaluate the basic elements of an arithmetic expression `AE[T]` is defined:

```
trait AE_Eval[T] {
  abstract override def modify(ae : AE[T]) : Int = ae match {
    case AE_Plus(x : Int, y: Int) => x + y
    ...
    case sthelse => super.modify(sthelse)
  }
}
```

Let an arithmetic expression be extended by an atomic elements `AE_Value`, a further trait can be defined which performs the evaluation of the atomic element `AE_Valued`:

```
trait AE_EvalValued[T] {
  abstract override def modify(ae : AE[T]) : Int = ae match {
    case AE_Value(x : Int) => x
    case sthelse => super.modify(sthelse)
  }
}
```

Let the extended arithmetic expression be characterised by denoting the type variable `T` as AEV, an evaluator can be easily instantiated by extending the traversal class by the standard evaluation and the extended evaluation:

```
new AE_Transform[AEV,Int] extends AE_Eval[AEV] with AE_EvalValued[AEV]
```

4.2.2 LARES-related Formalisms & Transformations

Following the formal definition of the traversal function in Section 3.1, in sub-project `ftse-lares` an abstract traversal trait is defined which is responsible to traverse a LARES model along its instances. Thereby, FW and BW define abstract type parameters which represent data structures that are used to forward information towards the leaves or to relay information towards the root node during the traversal of the instance tree:

```
trait LaresTraverser[BW,FW]
```

Two important abstract methods are declared. At the one hand, the forwards method to propagate information from an arbitrary intermediate node towards their leaves is declared:

```
def forwards(i : Instance, fw : FW) : (Instance,FW)
```

It contains the parameter i for the visited instance and the forward information parameter fw of type FW obtained from the parent instance. The visited instance i and the forward information are processed and returned.

On the other hand, the backwards method is declared. It also has an instance parameter i and a forward information parameter fw. It further includes an additional structure bw which represents the backward information calculated from the processed child instances:

```
def backwards(ri : Instance, fw : FW, bw : Iterable[BW]) : BW
```

The backwards method is responsible to process the visited instance i by taking the backward information bw and the forward information fw into account.

Finally, the recursive traverse method is defined. It is initially applied to the root instance of a LARES specification and uses the forward information fw to determine the subsequent forward information fwres. Then all subinstances are recursively processed in order to calculate the backward information which consists of the processed subinstances. Finally, the backwards method is called to process the currently visited instance and to calculate the backward information therewith:

```
def traverse(i : Instance, fw : FW) : BW = {
  val fwres = forwards(i,fw)
  backwards(
    fwres._1, fwres._2,
    fwres._1.body.instances.map(child => {
      traverse(child,fwres._2)
    })
  )
}
```

Note that _1 and _2 denote projections of the first and the second element of a tuple respectively.

The abstract traversal trait LaresTraverser has to be mixed in the traits which implement the different transformations as detailed in the subsequent sections. For each specific transformation, the forward and backward data structure types as well as the implementation of the methods according the transformations semantics are sketched.

Parameter Evaluation and Instance Tree Construction

Concerning the parameter transformation, the backward and forward data structures are defined within a parameter expansion structure object PES:

```
object PES {
  type BW = Instance
  type FW = (
    Map[Identifier,Identifier],
    HashMap[Identifier,LBehavior],
    HashMap[Identifier,ModuleDefinition]
  )
}
```

The transformation to evaluate parameters and to thereby resolve expand statements for the purpose of constructing the instance tree is implemented by the class Instantiation which, as mentioned in the previous section, inherits from the trait LaresTraverser parametrised by the types defined inside of PES. Furthermore, the transformation interface AbstrTransformer is inherited:

```
class Instantiation extends
  LaresTraverser[PES.BW,PES.FW] with
  AbstrTransformer[LARES_Element, LARES_Element] with ...
```

Inheriting from AbstrTransformer and LaresTraverser requires implementing the transform, forwards and backwards method.

If transform matches the root element of a LARES specification (SpecLares), the forward information which contains the scope of visible definitions (when regarding this transformation) will be defined by the therein available Behavior and Module definitions. When the traverse method is executed, it returns the backward information with the processed System instance which is subsequently embedded in the LARES specification element and returned by the transform method:

```
def transform(e : LARES_Element) : LARES_Element = e match {
  case s : SpecLares => {
    val B = HashMap((for (b<-s.behaviors) yield b.identifier->b).toSeq :_*)
    val M = HashMap((for (m<-s.modules) yield m.identifier->m).toSeq :_*)
    val fw = (HashMap[Identifier,Identifier](), B, M)
    SpecLares(List(),List(),traverse(s.system,fw))
  }
}
```

- The `forwards` method has to be overwritten in order to implement the evaluation of parameters and the resolution of the thereby dependent statements:

  ```
  override def forwards (i:Instance, fw:FW) = { ... }
  ```

  Its implementation is not further detailed, as it follows the formal semantics from Section 3.3.1. It is only sketched by different steps:

  1. Firstly, the Module definitions *in scope* are extracted from the forward information to determine the definition corresponding to the current instantiation.
  2. Subsequently, the parameters for the instantiation (which are either defined by the instantiation or by the default values of the Module definition) are determined.
  3. Then, the Module definition's body is resolved such that all contained statements (including expand statements) are resolved using the evaluated parameters.
  4. Subsequently, the local Behavior and Module definitions become part of the scope and will hence be included in the forward information.
  5. Moreover, the set of delegates is subdivided regarding the criterion whether a delegate addresses a Behavior or Module definition. Those which address a Module definition can be handled as Instances and hence be appended to the instances of the resolved body.
  6. Furthermore, the addressed Initial statements have to be determined by forward information in order to obtain the initial configuration of a Module instance. The first Initial specified will be taken, if no corresponding reference could be determined from the forward information. The resolved Behavior instances are constructed by the evaluated parameters and the initial configuration.
  7. Finally, the resolved Module instance is constructed (from the identifier, the evaluated parameters, the Behavior instances and the new Module body) and returned together with the forward information.

- The `backwards` method replaces subinstances with their processed version obtained by the backward information `bw`:

  ```
  override def backwards (i:Instance, fw:FW, bw:Iterable[BW]) = { ... }
  ```

Apart from the standard language elements of LARES, auxiliary statements can be defined for further extensions. In order to take this into account, the stackable trait pattern is employed again to decouple the different extensions from the standard transformation. Therefore, a trait `AbstractLaresParameterResolver` has been defined in order to resolve the auxiliary statements:

```
trait AbstractLaresParameterResolver {
  def resolveAuxStatements(
    auxStmts : List[ModuleStatement],
    evaluations : HashMap[Identifier,Either[Int,Double]]
  ) : List[ModuleStatement]
}
```

The method `resolveAuxStatements` is responsible to evaluate the variables of all auxiliary statements considering the given evaluations. The specific implementations for the base class and all stacked traits for further extensions are not detailed here, as they follow the formal transformation semantics described in Section 3.3.1 or can be looked-up in the implemented code.

Condition Statement Transformation

In order to resolve the references to intermediate Condition statements, the addressed condition expressions have to replace the references. The transformation of Condition statements simply requires relaying the *resolved* conditions backwards. For this purpose, the data structure object `CES` has been defined to provide the desired `BW` type (of a processed instance which contains only resolved Condition statements):

```
object CES { type BW = Instance; type FW = Null }
```

Similarly to the description given in the previous section, the *condition expansion* transformation is implemented by the following class definition:

```
class ConditionRes extends
  AbstrTransformer[LARES_Element,LARES_Element] with
  LaresTraverser[CES.BW,CES.FW] with ...
```

Similarly to the previous transformation, the `forwards` and the `backwards` methods have to be implemented. As no information is required to be forwarded in the direction to the leaves, the implementation of the `forwards` method is straightforward and simply passes along the instance as it is.

In order to resolve all statements which contain condition expressions such as Condition, forward and guards statements, the local Condition statements have to first be resolved in the order of their interdependencies. As a result, all local references inside a condition expression can be resolved, because all referenced local Condition statements can be resolved due to the given order. All non-local references can be resolved anyway by using the resolved Conditions of each child Module instance obtained via the instances inside the backward information or from State statements addressed via Behavior instances.

The resolution process substitutes the references by the addressed (visible) condition expressions. It thereby implicitly implements the scope of a Condition statement or a State statement, such that a named statement is in general either visible locally (within the defining instance) or from the direct environment (i.e. the child instances). After resolving all local Conditions, the forward and the guards statement can be resolved by additionally taking the resolved local Condition statements into account.

Condition expressions may also be contained in auxiliary statements other than Condition, forward and guards statements. For this case, the stackable trait pattern is applied in order to decouple the resolution process of the language elements which contain condition expressions. Each (stacked) trait which inherits from AbstrConditionExpansion must have a specific implementation of the method resolveAuxConditionStatements to resolve the auxiliary statements by the referred condition expressions:

```
trait AbstrConditionExpansion {
  def resolveAuxConditionStatements(
      aux : List[ModuleStatement],
      resRemote : Iterable[(Identifier,Condition)], // non-local condition
      resLocal : Seq[Condition] // local condition
  ) : List[ModuleStatement]
}
```

Based on resolved statements, backwards returns the backward information containing the processed instance and therein the resolved local Conditions.

forward Statement Transformation

The forwards transformation is similar to the resolution of condition expressions. Again no information is required to be forwarded towards the leaf nodes. Instead of resolved condition expressions, resolved forwards statements are relayed backwards wrapped inside the processed instances:

```
object FES { type BW = Instance; type FW = Null }
```

Consequently, the transformation class inherits the same traits, i.e. from AbstrTransformer and LaresTraverser. The latter is parametrised with the initially defined data structure types FW and BW:

```
class ForwardRes extends
  AbstrTransformer[LARES_Element, LARES_Element] with
  LaresTraverser[FES.BW,FES.FW] with ...
```

- The `forwards` method simply passes the instance along, as no further information has to be propagated towards the leaf nodes

  ```
  override def forwards(i:Instance, fw:FW) = (i,null)
  ```

- In contrast, the `backwards` method is responsible to resolve the given instance `i` and the forward information `fw` (which might be addressed by the parent Module instance):

  ```
  override def backwards(i:Instance, fw:FW, bw:Iterable[BW]) = { ... }
  ```

In order to determine the set of visible forward statements which may be addressed via their guard labels by some local reactive expression, the backward information is used to extract the forward statements of the contained child instances and the guarded transitions of the instantiated Behaviors (which can be regarded as resolved forwards).

Note that the type of a resolved reference to a guard label (beside the standard reference information) is complemented by a distribution type. It is applied when a guards statement is resolved in order to be able to check whether the synchronous reaction arising thereof is composed of transitions which have the same distribution type. This check is not performed in the forward statement transformation, as this is not calculated on the reactive expression, but for each product term which arises thereof individually when applying the transformation to derive the SPA or the LTS.

Subsequently, all local forward statements are processed (i.e. such that the referenced forwards can be substituted) following the topological order of their interdependencies. Due to the conditions assigned to reactive expressions and the combinatorics arising from choices in general, new choices can arise for a forward label as described in Section 3.3.3 and implemented according to that.

A consecutive application of all transformations introduced up to this step will construct the LARES$_{\text{BASE}}$ representation of the model. According to the subsequent transformations, different target models are generated which can be addressed by corresponding solvers.

SPA Transformation

This section focuses on the CASPA stochastic process algebra solver as the addressed tool. The meta-model of the CASPA SPA is again defined by *case classes*. Hereby each language element is derived from SPA_Element. As a consequence, a transformation trait can be defined which is parametrised by LARES_Element as source model type and SPA_Element as target model type:

```
class LaresSPATransformer extends
  AbstrTransformer[LARES_Element,SPA_Element] with
  LaresTraverser[GEN_SPA_TYPES.BW,GEN_SPA_TYPES.FW] with
```

The above used type parameters for the forward and the backward information are defined inside GEN_SPA_TYPES:

```
object GEN_SPA_TYPES {
  type FW = (Namespace,List[GenerativeLiteral],List[ReactiveLiteral])
  type BW = PACT_Element
}
```

The forward structure is a tuple consisting of the namespace of the Module instance which forwards the information, a list of propagated generative literals and a list of reactive literals. The above traversal class contains the implementation which finally transforms Behavior instances into sequential processes by taking all obtained generative and reactive literals into account, i.e. a process algebra tuple PAT which includes the sequential SPA process and synchronisation information which relates to the labels generated due to generative and reactive literals of product term combinations. A PAT element is derived from an abstract class PACT and hence represents a leaf element of a PACT structure, whereas the process algebra composition PACN element is derived from PACT as well, but is generated for each Module instance. In order to construct these structures, the forwards and the backwards method have to be implemented:

- The forwards method determines which generative/reactive literals have to be constructed and additionally forwarded for a given instance. Note that each literal is augmented with additional information such as a namespace, an encoding, a Boolean interaction flag (which states whether only a single instance is addressed, i.e. leading to an action whose label may not become part of the synchronisation set) and a common distribution type. The helper function filters the relevant literals for the given instance. It modifies all *conditional reactive*s by combining their generative expressions with the condition of corresponding guards statement and collects them. They are encoded by a unique number. Each of the resolved *conditional reactive*s is checked whether a common distribution can be calculated. Then for both the generative and the reactive part, Binary Decision Diagrams (BDD) will be constructed in order to determine the generative product terms and the reactive product terms, respectively, if the generative expression is satisfiable (i.e. the root node of the BDD is not the Boolean false terminal). Product terms which would require a single instantiated Behavior to be in two different states at the same time are not satisfiable and are therefore removed. The sets of product terms are subsequently enumerated and combined such that augmented literals can be built.

- The backwards method performs the SPA transformation. For that purpose, it groups the obtained generative and reactive literals by behaviour instance identifiers, and calls the transformation method for all behaviour instances with their matching literal group as argument. The transformation implementation of a Behavior instance follows exactly the formal semantics denoted in 3.5.4. The constructed sequential processes are complemented by the synchronisation information (i.e. the process algebra tuple (PAT) structures) and then composed by constructing the process algebra composition tuples (PACT) structures.

Following the composition semantics of 3.5.3, the SPA specification can be generated in a subsequent transformation step and pretty printed in order to obtain the textual SPA specification which can finally be fed into the CASPA solver.

LARES$_{\text{FLAT}}$ Transformation

If the target model is planar, the hierarchy will not be preserved by any means. Consequently, a further transformation is defined which is responsible for resolving the hierarchy of a LARES model. For this reason, the LARES$_{\text{FLAT}}$ Expansion Structures (LFES) defines the forward information (which carries the current namespace and the guards statements augmented by information for unique identification) and the backward information (which consists of LARES$_{\text{FLAT}}$ automata, guards and measure statements):

```
object LFES {
  type FW = (Namespace,List[(Namespace,Int,LGuard)])
  type BW = (List[LFA],List[LGuard],List[Measure])
}
```

In contrast to the SPA transformation, the LARES$_{\text{BASE}}$ specification is now transformed into a flattened model denoted as LARES$_{\text{FLAT}}$. The data type of the LARES$_{\text{FLAT}}$ formalism is named LFIA which describes a system of interacting automata. Accordingly, the transformation class is given as follows:

```
class L2LFGen extends
  AbstrTransformer[LARES_Element,LFIA] with
  LaresTraverser[LFES.BW,LFES.FW] with ...
```

Again, the transformation inherits from the traversal trait which requires implementing the `forwards` and `backwards` methods:

- The `forwards` method first updates the namespace (by considering the processed Module instance) and then adapts the namespace of each local guards statement. The adapted local guards and the guards statements from the forward information are unified in order to construct the forward information for the currently processed module instance.

- The `backwards` method collects the information relayed backwards from the child instances (e.g. the constructed automata, the adapted guards statements and measures). Firstly, each local Behavior instance of the currently processed Module instance is translated such that it is also complemented by the current namespace. The $\text{LARES}_{\text{FLAT}}$ model (`LFA`) is built straightforwardly by including the states, the guarded as well as unguarded transitions of a Behavior instance, and the instance identifier which is complemented by the given namespace. Secondly, the local guards are processed (as well as the measures). The guards statements and the `LFA` instances are then aggregated and returned as part of the backward information of the currently processed Module instance.

$\text{LARES}_{\text{FLAT}}$ to Labelled Transition System

A reachability algorithm on $\text{LARES}_{\text{FLAT}}$ implements the LTS semantics as given in 3.4.3. The base class of the transformation inherits from `AbstrTransformer` which is parametrised by the types `LFIA` as input model type and `TRA` as output model type:

```
class LF2TS extends
  AbstrTransformer[LFIA,TRA] with ...
```

The `transform` method implements a (breadth/depth first) search to exploit the reachable state space. Firstly, the initial states of all `LFA` instances serve as a composed initial state for the fixed point algorithm which is performed as long as further states can be explored. As described in Section 3.4.3, guards statements are used to determine whether the current composed state satisfies a generative condition, and the set of currently available guarded transitions is used to decide whether a reactive part is satisfied. Accordingly, a number of synchronous composed transitions into reachable composed target states can be determined. All transitions which have been traversed while exploring the state space are stored in the reachability graph which is also known as the transition system (TRA).

Composition of Transformations

Since every transformation is derived from `AbstrTransformer` and implements the transform method, complex multi-step transformations can be composed in Scala:

```
def lares2lbase =
  ((new Instantiation with InstantiationReward) transform) _ andThen
  ((new ConditionRes with ConditionResReward) transform) _ andThen
  ((new ForwardRes) transform) _
```

The above composition has two dimensions of extensibility: On the one hand, additional consecutive steps can be appended (e.g. to address further solvers). On the other hand, also a transformation can be extended using the stackable trait principle in order to deal with language extensions. Due to the chosen approach, additional transformation steps can be defined independently and enriched with further stackable traits.

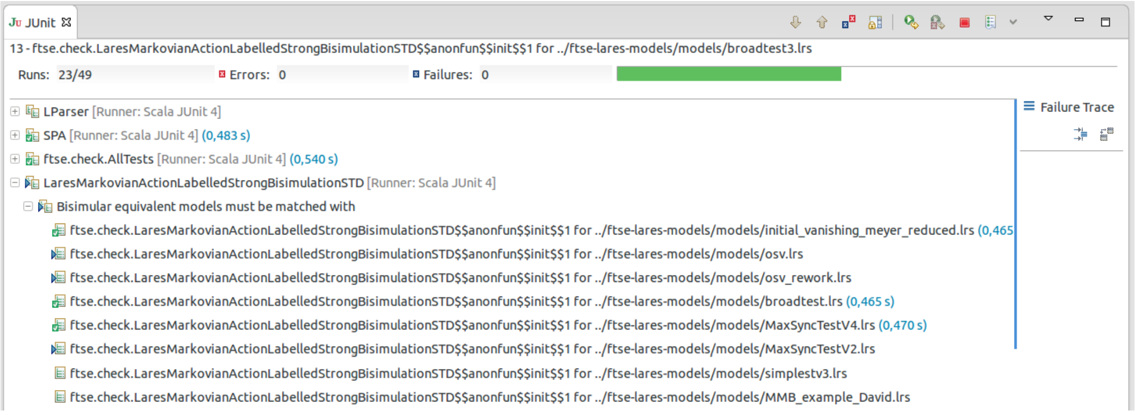
The transformation of the new statements `StateReward` and `TransitionReward` as part of the *reward extension* of the LARES language serves as example. As a second example, the extension by non-deterministic actions may be quoted. These language extensions can be found in the `ftse-lares-extensions` sub-project (not shown in Figure 4.5) following the theoretical work of [69, 70] and its implementation described in [101]. For both extensions, the additional concepts added to the LARES language and the semantics to perform the transformations to their specific target languages are not subject of this thesis and thus also their implementation is not detailed here.

4.2.3 LARES Model Transformation Validation

The sub-project `ftse-lares-check` is responsible for maintaining validity of the transformation semantics during ongoing development (improving code, extending languages and transformations, etc.). There, the *specs*^{2}-framework has been used [134]. A number of tests can be performed in order to check whether the parsers work correctly with a set of given test-models. Moreover, bisimulation equivalence of two kinds of transformations can be performed. This means that two TRA objects are compared for bisimilarity by

1. transforming LARES models to a CASPA stochastic process algebra (such that the reachability execution performed by the CASPA tool yields the first TRA object), and by

2. performing reachability analysis on a LARES$_{\text{FLAT}}$ representation (which yields the other TRA object).



```
13 - ftse.check.LaresMarkovianActionLabelledStrongBisimulationSTD$$anonfun$$init$$1 for ../ftse-lares-models/models/broadtest3.lrs
Runs:  23/49          Errors:  0          Failures:  0

⊞ LParser [Runner: Scala JUnit 4]
⊞ SPA [Runner: Scala JUnit 4] (0,483 s)
⊞ ftse.check.AllTests [Runner: Scala JUnit 4] (0,540 s)
⊟ LaresMarkovianActionLabelledStrongBisimulationSTD [Runner: Scala JUnit 4]
  ⊟ Bisimular equivalent models must be matched with
      ftse.check.LaresMarkovianActionLabelledStrongBisimulationSTD$$anonfun$$init$$1 for ../ftse-lares-models/models/initial_vanishing_meyer_reduced.lrs (0,465
      ftse.check.LaresMarkovianActionLabelledStrongBisimulationSTD$$anonfun$$init$$1 for ../ftse-lares-models/models/osv.lrs
      ftse.check.LaresMarkovianActionLabelledStrongBisimulationSTD$$anonfun$$init$$1 for ../ftse-lares-models/models/osv_rework.lrs
      ftse.check.LaresMarkovianActionLabelledStrongBisimulationSTD$$anonfun$$init$$1 for ../ftse-lares-models/models/broadtest.lrs (0,465 s)
      ftse.check.LaresMarkovianActionLabelledStrongBisimulationSTD$$anonfun$$init$$1 for ../ftse-lares-models/models/MaxSyncTestV4.lrs (0,470 s)
      ftse.check.LaresMarkovianActionLabelledStrongBisimulationSTD$$anonfun$$init$$1 for ../ftse-lares-models/models/MaxSyncTestV2.lrs
      ftse.check.LaresMarkovianActionLabelledStrongBisimulationSTD$$anonfun$$init$$1 for ../ftse-lares-models/models/simplestv3.lrs
      ftse.check.LaresMarkovianActionLabelledStrongBisimulationSTD$$anonfun$$init$$1 for ../ftse-lares-models/models/MMB_example_David.lrs
```

Figure 4.6: Unit testing the LARES transformation implementations (screenshot)

Figure 4.6 shows an iteration through all test models which are used to check the different parsers and transformations. It is important that LARES test models are designed such that the test-coverage is as large as possible. Therefore, aspects related to the combinatorics originating from the product terms and the choices between the generated transitions are captured by different models which implement variants of forward and guards statements.

Chapter 5

Case Studies and Implications on Scalability

In order to demonstrate the applicability of the LARES approach, some models have been developed to show that

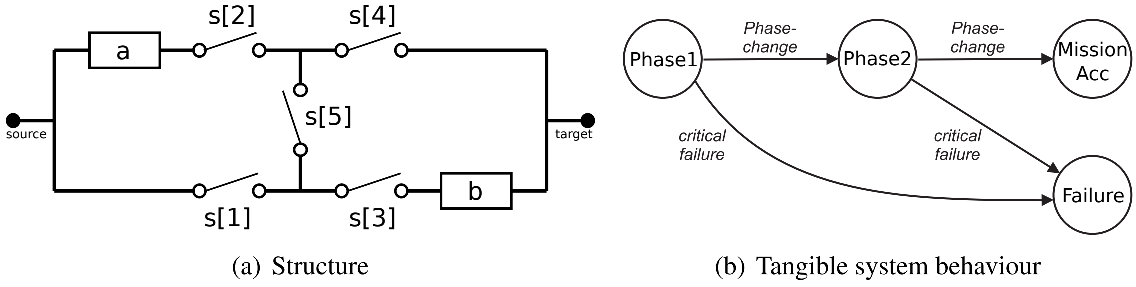
- the required dependability aspects can be modelled,
- modellers can access the language easily,
- the model transformation scales with respect to the input model and
- the whole analysis process is carried out smoothly using the provided LARES toolset.

Therefore, Section 5.1 picks up the Phased Mission System (PMS) case-study discussed in [68] to do the still pending analysis. In Section 5.2, a larger real-world case study to determine the dependability of the spring and tilt module of a rail vehicle is recalled (RailCab). Section 5.3 by contrast defines a new parametrisable, recursively layered queueing network. Finally, Section 5.4 determines metrics for the LARES modelling language by considering the above models including the Fault Tolerant Network and the Component Monitoring System model of Section 1.2 and 1.3 respectively. The resulting SPA model of the transformation and the MTBDDs (constructed by the CASPA tool) are taken into account to evaluate scalability aspects of the LARES approach.

5.1 Modelling and Analysis of a Phased Mission System

The Phased-Mission-System (PMS) model originally presented in [30] was manually transformed into the CASPA SPA [124]. Furthermore, it was used as a basis to show the expressiveness of the LARES language by formulating a more complex version thereof

[68]. The PMS LARES model is now taken up again in order to catch up on the analysis. In this work, the LARES toolchain is utilised to apply the described transformations which automatically construct SPA models from high-level LARES specifications and to evaluate measures of interest along with the CASPA tool.



(a) Structure

(b) Tangible system behaviour

Figure 5.1: PMS (compiled from [123, p. 123])

The PMS system consists of two components `a` and `b`, five switches `s[i]` and conductors which interconnect the components and the switches. The goal of the system is to realise a connection between the source and the target node as shown in Figure 5.1(a). The figure can be regarded as an RBD in which the system will be considered to be working if at least one connection between source and target is realised. Two consecutive phases which are prone to failure have to be performed before the system's mission is fulfilled (cf. Figure 5.1(b)). The initial phase is realised by switching positions which establish a redundant parallel operation of both components a and b. Once the first phase is finished, the system reconfigures itself to a serial structure requiring both components to operate. Within both phases, the system may fail depending on the operational availability of the components and the positions of the switches to realise a connection.

In order to specify the above description of the system, the system Behavior definition consists of two states which represent the consecutive phases which end up in either a state representing the accomplishment of both phases or a failure. Note that the state of reconfiguration `Reconf` does not appear in Figure 5.1(b) since it is vanishing and gets eliminated. The duration of each phase is determined by an exponential distribution with rate `tau1` or `tau2` given as parameters. The system's failure in both phases might be externally triggered by providing the guarded transitions labelled with `Fail_P1` or `Fail_P2`.

```
Behavior B_Phases(tau1, tau2) {
   State Phase1, Reconf, Phase2, Failure, Mission_Acc
   Transitions from Phase1
     if ⟨Fail_P1⟩ → Failure
                  → Reconf, delay exponential tau1
   Transitions from Reconf
```

```
    if ⟨switch⟩  → Phase2
  Transitions from Phase2
    if ⟨Fail_P2⟩ → Failure
                 → Mission_Acc, delay exponential tau2
}
```

The system itself consists of a number of switches that are arranged such that a connection can be realised between the source and the target node. The Behavior definition of a switch comprises the states with the self-evident identifiers Open and Closed including the error states stuck-at-open (SA_Open) and stuck-at-closed (SA_Closed). Two guarded transitions for the states to intentionally close a switch int_close (leading to either Closed or to SA_Open) or to intentionally open a switch int_open (leading to either Open or SA_Closed) are defined. The transition which is eventually taken depends on the ratio given by the weights gamma and 1 − gamma. In addition, a switch may unintentionally open with regard to an exponential distribution such that the erroneous state SA_Open is reached.

```
Behavior B_Switch(gamma, MTTF) {
  State Closed, Open, SA_Open, SA_Closed
  Transitions from Open
    if ⟨int_close⟩  → Closed, weight 1 − gamma
    if ⟨int_close⟩  → SA_Open, weight gamma
  Transitions from Closed
    if ⟨int_open⟩  → Open, weight 1 − gamma
    if ⟨int_open⟩  → SA_Closed, weight gamma
                   → SA_Open, delay exponential 1.0/MTTF
}
```

A component may also fail. A Boolean failure Behavior definition therefore describes an exponential distribution of a time delay specified by the parameter MTTF until the current state Active changes to Failed:

```
Behavior B_NonRepairable(MTTF) {
  State Active, Failed
  Transitions from Active
    → Failed, delay exponential 1.0/MTTF
}
```

The System definition captures the interaction behaviour between all defined subinstances (i.e. switches and components) and the inherited system behaviour. It specifies the measures of interest and it may provide further Module (as in this case) or Behavior definitions:

```
System PMS : B_Phases(tau1=1.0/100, tau2=1.0/50) {
  /** Module−Type Definitions **/
  Module M_Component: B_NonRepairable(MTTF=10000) { ... }
  Module M_Switch: B_Switch(gamma=0.05,MTTF=10000) { ... }

  /** Submodule-Instances **/ ...
  /** Interaction Behaviour **/ ...
  /** Measures **/ ...
}
```

The Module M_Component definition inherits from B_NonRepairable setting the parameter MTTF to 10000 hours. It provides the Initial statement Active and reveals its failed state via the Condition statement Failed to the environment.

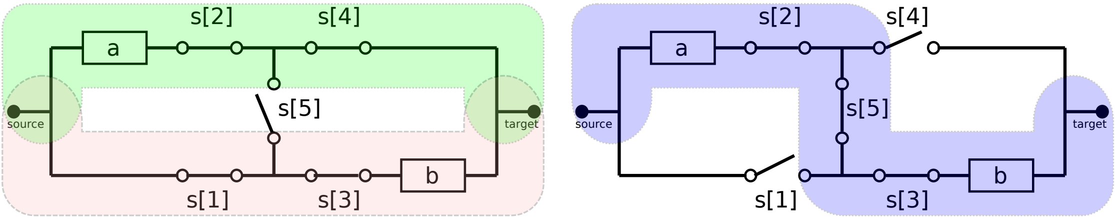
```
Module M_Component: B_NonRepairable(MTTF=10000) {
  Initial Active = B_NonRepairable.Active
  Condition Failed = B_NonRepairable.Failed
}
```

The Module definition M_Switch inherits from B_Switch and sets the parameters for gamma and MTTF. Two Initial statements are provided. These allow setting a switch to be closed or opened initially. Two Conditions additionally state whether a switch is closed or stuck-at-closed. Two forward statements are defined which both deliver the int_open and < int_close > event to the internal behaviour in order to intentionally open and close a switch respectively:

```
Module M_Switch: B_Switch(gamma=0.05,MTTF=10000) {
  Initial InitClosed = B_Switch.Closed
  Initial InitOpen = B_Switch.Open
  Condition Closed = B_Switch.Closed
  Condition SA_Closed = B_Switch.SA_Closed
  forward ⟨int_open⟩ to B_Switch.⟨int_open⟩
  forward ⟨int_close⟩ to B_Switch.⟨int_close⟩
}
```

The System definition further specifies subinstantiations. In order to specify the initial configuration of the first phase, four of the five instantiated switches are configured such that they are initially closed whereas the last one remains open. Furthermore, both instantiated components are initially active.

```
/******** Submodule-Instances ********/
expand (cld in{1 .. 4}) { Instance s[cld] initially InitClosed of
    M_Switch }
Instance s[5]  initially InitOpen of M_Switch
Instance a initially Active of M_Component
Instance b initially Active of M_Component
```



(a) Phase1 (b) Phase2

Figure 5.2: Phases of the PMS

The PMS is constructed in such form that it should cut off a failed component from a connection by opening the associated switch. The System definition captures that interaction behaviour by the first two guards statements.

```
/************* Interaction Behaviour *************/
a.Failed guards s[2].⟨int_open⟩
b.Failed guards s[3].⟨int_open⟩
```

The first phase will run as long as there is at least one of the connections associated with component a including the switches s[2] and s[4] or associated with component b including s[1] and s[3] established (cf. Figure 5.2(a)). Otherwise, the phase fails which will trigger the system behaviour via Fail_P1.

```
( (a.Failed | not s[2].Closed | not s[4].Closed) &
  (b.Failed | not s[1].Closed | not s[3].Closed) )
  | s[2].SA_Closed | s[3].SA_Closed guards B_Phases.⟨Fail_P1⟩
```

If the system withstands the first phase without a failure, the system reconfigures in order to perform the second phase (cf. Figure 5.2(b)). Therefore, the system attempts to open the switches s[1] and s[4], to close switch s[5] and to synchronously shift into phase 2:

```
true guards sync { maxsync {s[1].⟨int_open⟩, s[4].⟨int_open⟩, s
    [5].⟨int_close⟩}, B_Phases.⟨switch⟩}
```

The second phase can be performed as long as none of the components fail and the switches s[2], s[3], s[5] are closed. Only then may the system accomplish its mission.

```
a.Failed | b.Failed | not s[2].Closed | not s[3].Closed |
  not s[5].Closed guards B_Phases.⟨Fail_P2⟩
```

The instance tree constructed for the PMS model is shown in Figure 5.3. Note that behaviour instances are represented by leaf nodes, so that each node is associated with its sequential behaviour definition. Each of the 5 switches has 4 states. At the same time each of the components a and b has two states and the system behaviour comprises 5 states.

Note that subtle notational changes accounted by the evolution of LARES are considered and applied to the model presented in [68]. Furthermore, this model fixes an issue

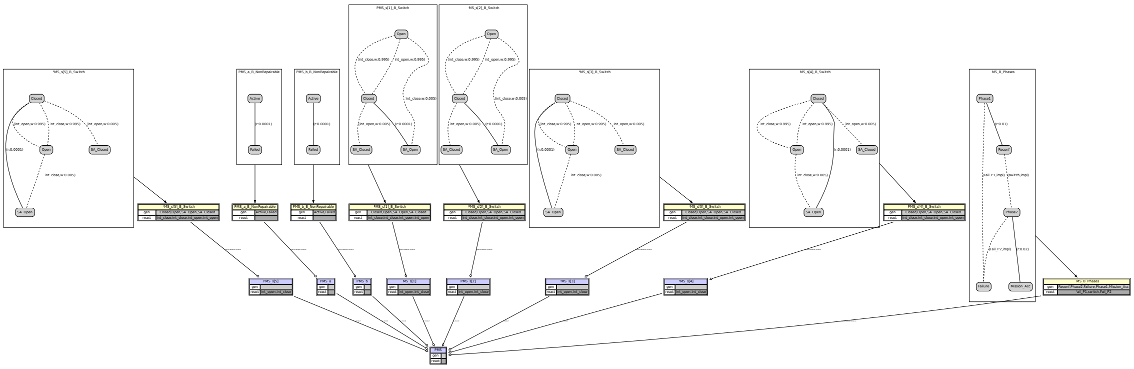


Figure 5.3: PMS instance tree (generated by the LARES toolset)

concerning the rates and weights (i.e. the rates were spuriously set by the mean time of certain events and the weighting gamma of an erroneous switching event in B_Switch was falsely set as the complement of gamma instead of $1 - \texttt{gamma}$). Lastly, the reconfiguration was primarily initiated when it entered the second phase. This led to an unintended behaviour as this procedure was not atomic and instead concurred with the failure condition of the second phase. It was resolved by adding an intermediate vanishing state to the system behaviour in which the switches are instantaneously reconfigured to the requirements of the next phase. Note that alternatively one would resolve the above by synchronising the reconfiguration with the Markovian transition from the first into the second phase without introducing an additional state.

In addition, certain measures of interest have been formulated via Probability statements in order to calculate the transient probability of each tangible state, i.e. Phase1, Phase2, MissionSuccess and MissionFailure:

```
/******** Measures ********/
Probability Phase1   = Transient(B_Phases.Phase1, 500)
Probability Phase2   = Transient(B_Phases.Phase2, 500)
Probability MissionSuccess   = Transient(B_Phases.Mission_Acc, 500)
Probability MissionFailure   = Transient(B_Phases.Failure, 500)
```

The results of the analysis are depicted in Figure 5.4. Since the system initially starts in Phase1, the probability at $t = 0$ is 1.0. If no failures occur, the sojourn time of the first phase will be determined by an exponential distribution. Then the system leaves Phase1 and enters Phase2. The state Reconf to reconfigure the switch positions is vanishing. As its sojourn time is always 0 and thus cannot aggregate any probability mass, it is eliminated before analysis. Whilst the probability of the first phase decreases, the probability mass over time thus accumulates in the second phase. The probability of the accomplishment of the mission (represented by the state PMS_MissionSuccess) simultaneously increases, due to the flow of probability mass from Phase2.

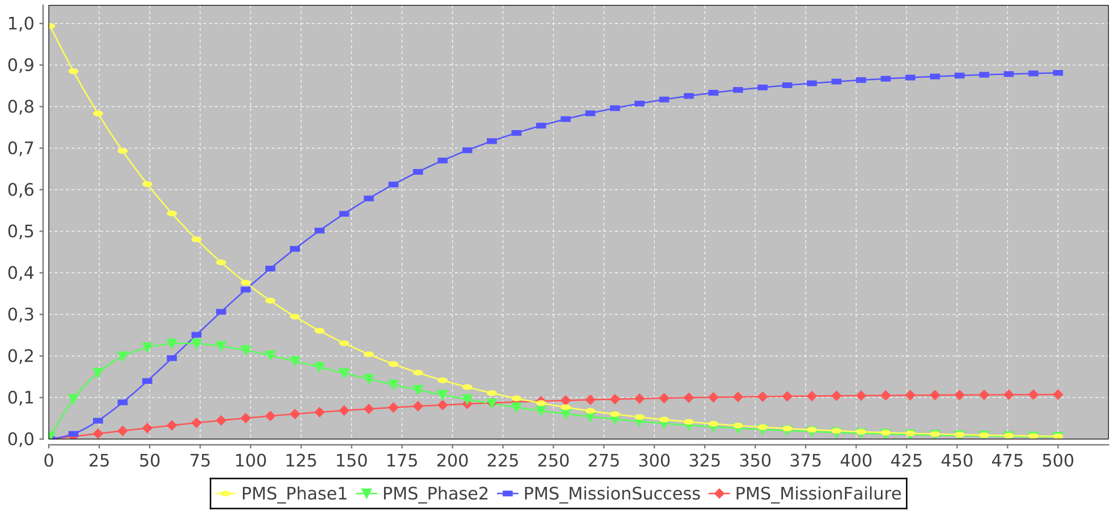


Figure 5.4: Transient analysis of state probabilities of the PMS model (generated by the LARES toolset)

Either because of on-demand-failures (which occur during the process of switching) or due to the MTTF of a switch or a component, the probability mass drains away towards the `Failure` state (`PMS_MissionFailure`) in each phase.

The probability of mission success heavily depends on the `MTTF` parameters of the components and the switches. Nonetheless, the parameter `gamma` also impacts the overall results. To emphasize this effect, further Probability statements have been defined (for selected representative behaviour instances as these behave symmetrically to the omitted ones, i.e. `s1` ∼ `s4` and `s2` ∼ `s3`) to highlight the probability of a switch to be either stuck-at-closed or stuck-at-open and the probability of a component to be failed:

```
Probability s1_SA_Open   = Transient(s[1].SA_Open, 500)
Probability s3_SA_Open   = Transient(s[3].SA_Open, 500)
Probability s5_SA_Open   = Transient(s[5].SA_Open, 500)
Probability s1_SA_Closed  = Transient(s[1].SA_Closed, 500)
Probability s3_SA_Closed  = Transient(s[3].SA_Closed, 500)
Probability a_failed   = Transient(a.Failed, 500)
```

Figure 5.5 depicts the results obtained from the failure measures defined beforehand. The probability of a component to fail at a certain time point successively increases. As the switches can fail from `Closed` to `SA_Open` defined by the MTTF, the initially closed switches `s`[1], `s`[2], `s`[3] and `s`[4] (represented by `s`[1] and `s`[3]) lose probability mass in favour of `SA_Open`. In contrast to the course of the probability results for `PMS_s3_SA_Open`, a stagnation can be observed for the results of `PMS_s1_SA_Open`. This can be explained due to the switching from `Closed` to `Open` when the reconfiguration for the second phase takes place. A further effect of the phase change becomes apparent in the probability results

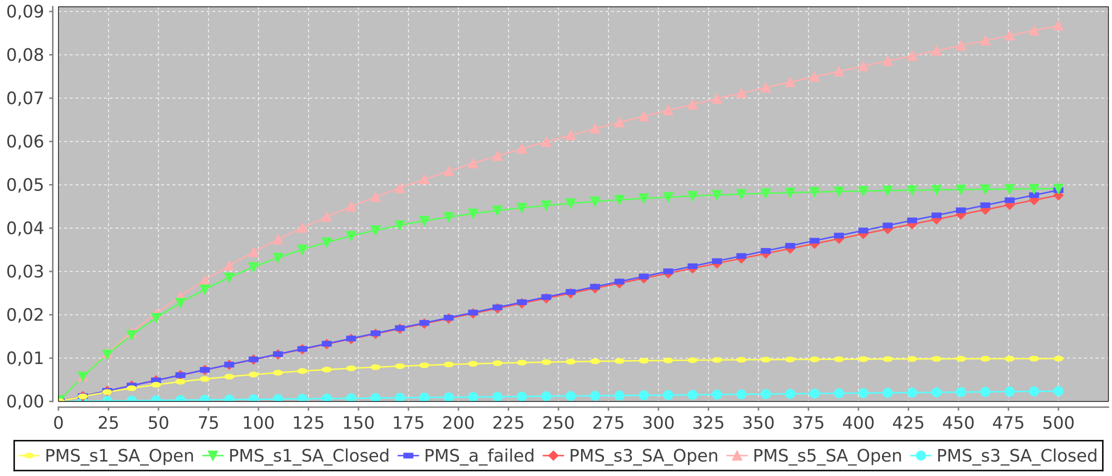


Figure 5.5: Transient analysis of representative PMS failure state probabilities showing the distribution of the overall probability mass (generated by the LARES toolset)

for `PMS_s1_SA_Closed`. It is here that the error-prone switching procedure determined by `gamma` comes to a gain of probability mass for `SA_Closed` which correspondingly to the end of the first phase stagnates. The measures obtained via `PMS_s5_SA_Open` aggregate probability mass by both the phase change and its MTTF. On the one hand this happens due to a failed closing procedure when the phase changes and on the other hand due to the transition from `Closed` to `SA_Open` following the MTTF. `PMS_s3_SA_Closed` will only be reached from `Closed` if it is switched. There is no switching of `s[3]` based on the phase change. If the associated component which in this case is `b` fails, this will be the only switching to take place. Due to the low probability of a component failure and the switching failure weight given by `gamma`, only a slight increase of probability mass can be observed over time.

5.2 Larger Case-Study: The RailCab Spring and Tilt Module

In [104] a real-world self-optimising mechatronical system [131] was modelled using LARES to determine its reliability. The complexity in terms of its instance tree can be perceived by Figure 5.6. A more detailed description including parts of the LARES model specification is given in [104]. The specified spring and tilt module is a functional submodule of a RailCab which also consists of modules for energy-supply, track-guidance, and drive-and-braking [97]. It seeks to achieve an almost complete vibration decoupling between the rail and the vehicle chassis by active suspension which is controlled by making use of sensor information about lateral and vertical disturbances by the rails. The use of an active suspension system in contrast to conventional dampers introduces further complexity to a system's behaviour. For this reason, the LARES language was chosen over other low-level approaches in order to determine the system's reliability by including the active suspension.

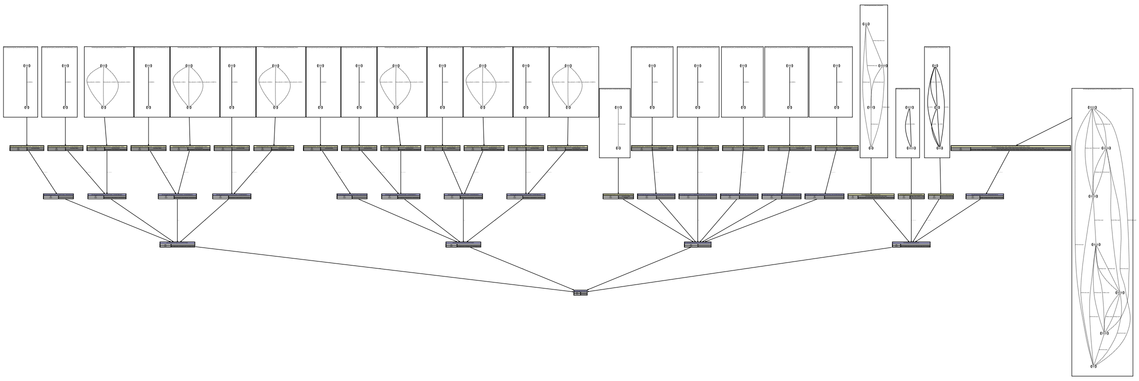


Figure 5.6: Instance tree of the RailCab model (generated by the LARES toolset)

Besides, also the usability of LARES as a modelling language could be explored and finally testified by the engineers who where responsible for developing the RailCab model (without having any prior knowledge of LARES). For this purpose, a first version of the LARES Editor was deployed. The modelling process has proven to benefit from the use of the editor because it directly supports the modellers in the act of specifying correct syntax. The requirements to the analysis workflow posed by the modellers had a big impact on the development of the LARES View-Plugin: The need to compare different measures within one 2D plot have been implemented and the parallel computation of transient analysis of a set of time-points exploited the available multi-cores and improved the analysis duration.

The RailCab model is described as follows. Acceleration sensors are represented as Boolean components, whereas the redundant cylinder modules [103] are non-Boolean: These are specified via two Behaviors, one to describe the inability to build up pressure in the cylinder and another one to describe an error in the control-loop which causes the cylinder to get stuck (at a known position).

The results of the analysis as shown in Figure 5.7 add validity to the behaviour expected by the modellers: After a few thousands of hours of operation the second level dominates, i.e. that some components have failed, but can still be stabilised. Afterwards, the third level becomes dominant, i.e. due to too many failures the system becomes unusable although it is still stable. There might also be failures that lead to a state in which the system is neither usable nor stabilisable and which will become increasingly dominant over time.

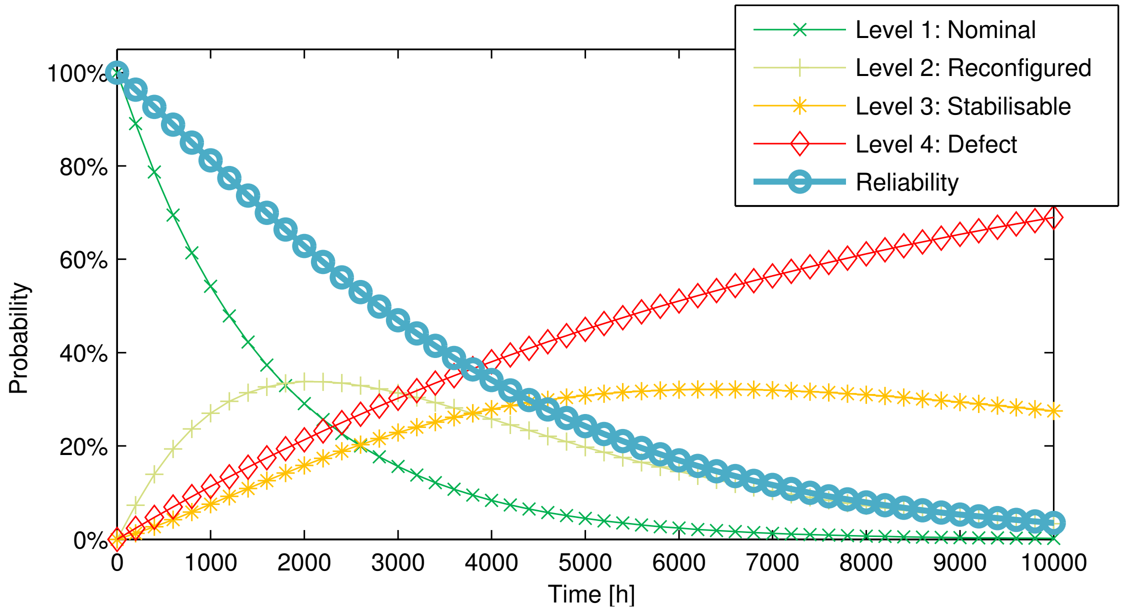
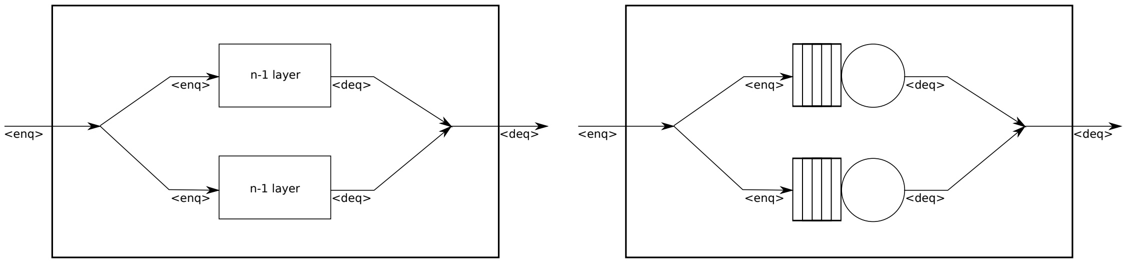


Figure 5.7: Analysis results of the RailCab model (compiled from [104])

5.3 Parametrisable Recursively Layered Queueing Network

As an incentive to be able to investigate the scalability of models, a layered queueing network (LQN) has been modelled that has a recursive structure. The parameters are evaluated while the model is instantiated. They determine its structure by decreasing the depth of the subtrees. An abstract queue layer consists of two other queues which might also be abstract layers depending on the recursion depth (see Figure 5.8(a)). The instantiation recursion terminates in a subtree when the parameter becomes 0. The concrete queue/processing unit is then instantiated as shown in Figure 5.8(b).



(a) n-level instance

(b) 0-level instance

Figure 5.8: Queue Layer Instances

The behaviour of a queue with length max can simply be modelled by defining states for the discrete fill level of the queue and their interrelating transitions, i.e. whenever a job is enqueued, the next state with an increased index will be entered and vice versa.

```
Behavior Q(max=1) {
  expand (l in {0 .. max}) { State stage[l] }
  expand (l in {0 .. max−1}) {
      Transitions from stage[l] if ⟨enq⟩ → stage[l+1]
      Transitions from stage[l+1] if ⟨take⟩ → stage[l]
  }
}
```

The processing behaviour consists of four states: idle means that a new job can be processed, processing denotes that a job is currently processed, processed means that the processing is finished, but the job has not yet been dequeued, and failed is a failure state which may occur while processing. Whenever a job can be processed, the transition into processing will be performed if <doProcess> is triggered. It may either take just the time defined by an exponential distribution with rate p until the job is processed, or the processing may fail following the Markovian rate f. A failure situation can be resolved following the Markovian rate rep. Once a job is processed and the transition guarded by deq is triggered, the behaviour enters the idle state again.

```
Behavior P(p=1.0, f=0.1, rep=0.5) {
   State idle, processing, processed, failed
   Transitions from idle if ⟨doProcess⟩ → processing
   Transitions from processing → processed, delay exponential p
                               → failed, delay exponential f
   Transitions from processed if ⟨deq⟩ → idle
   Transitions from failed → processing, delay exponential rep
}
```

A queueing unit comprises a queueing behaviour and a processing behaviour. Of course the enq and deq can be triggered by the environment, they are internally forwarded to the queue behaviour and the processing behaviour, respectively. The internal handover of a job is specified by a guards statement which synchronises the behaviours for taking a job from the queue and assigns the processing behaviour to do the processing.

```
Module U(qLength=1, pRate=1.0) : Q(max=qLength), P(p=pRate) {
         forward ⟨enq⟩ to Q.⟨enq⟩
         forward ⟨deq⟩ to P.⟨deq⟩
         true guards sync{Q.⟨take⟩, P.⟨doProcess⟩}
}
```

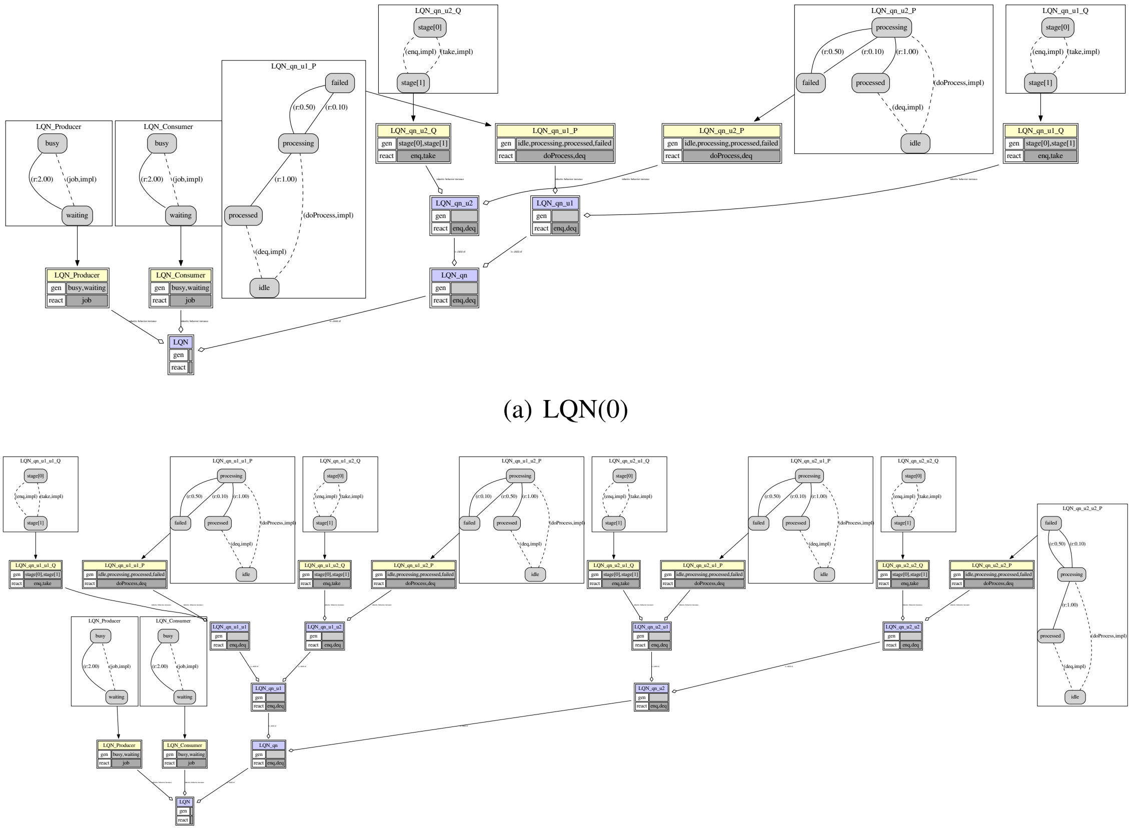
An important part of the model (corresponding to Figure 5.8(a) and 5.8(b)) is the Module definition of a queueing layer. There, a parameter denotes the layer number which is internally used to decide whether to terminate the recursion by instantiating a concrete queueing unit or to instantiate another layer. For this purpose, the expand statements are used to mimic conditional control structures as known from programming languages. When a Module definition UL is instantiated with parameter n = 0, the intersection of the sets {n} ∗ ∗{0} evaluates to a set with a single element, whereas the second statements {n} − −{0} determines the difference which in this case evaluates to the empty set. As a result, only the statements inside the first expand statement become effective. If n > 0 the evaluation will result the complement such that the statements of the second expand statements become effective. Internally queued jobs are probabilistically forwarded to the u1 or the u2 queue, expressed by a choice between two reactive expressions for each of the two cases.

```
Module UL(n) {
  expand (i in({n} ** {0})) /* leaf layer */ {
    Instance u1 of U
    Instance u2 of U
  }

  expand (i in({n} -- {0})) /* non-leaf layer */ {
    Instance u1 of UL(n=n-1)
    Instance u2 of UL(n=n-1)
  }

  forward ⟨enq⟩ to { u1.⟨enq⟩ u2.⟨enq⟩ } /* choice! */
  forward ⟨deq⟩ to { u1.⟨deq⟩ u2.⟨deq⟩ } /* choice! */
}
```

A simple worker behaviour is defined to be used for arbitrary delayed events following the Markovian parameter rate. The worker has two states, i.e. either being busy with a sojourn time given by the exponential distribution with the parameter rate or waiting as long as the next job arises.

```
Behavior Worker(rate) {
   Transitions from busy → waiting, delay exponential rate
   Transitions from waiting if ⟨job⟩ → busy
}
```

The Worker is used for both the production and the consumption of jobs. The System definition is specified such that it inherits these behaviours to produce the jobs to be enqueued in the layered queueing networks and to consume these jobs when they will be processed and hence dequeued. The interaction among the producer or the consumer



(a) LQN(0)

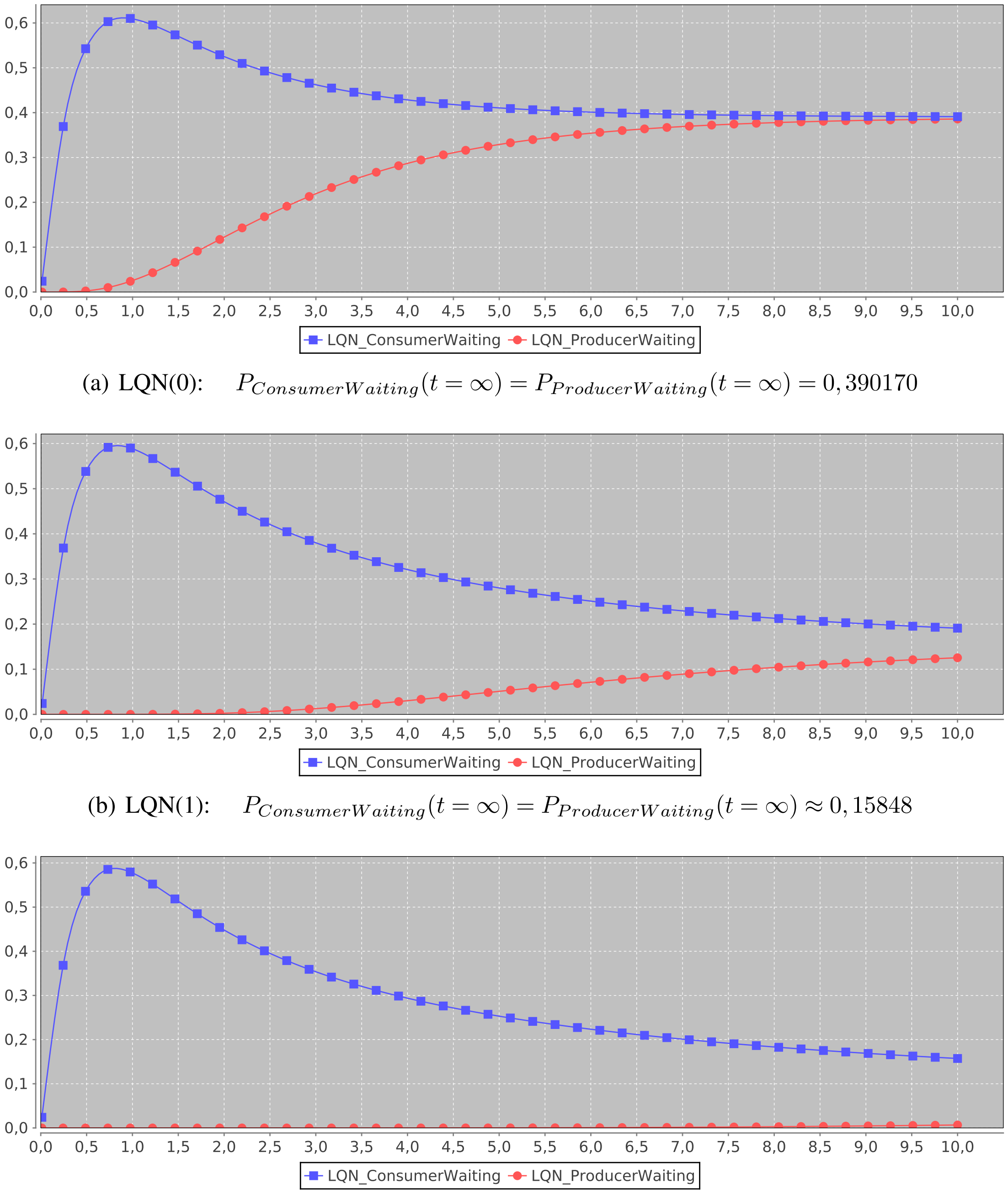
(b) LQN(1)

Figure 5.9: Parameter dependent instance trees (generated by the LARES toolset)

with the queueing system is specified by the two guards statements, as listed subsequently. Measures of interest are defined for this system by the Probability statements which consider the waiting time of both the consumer and the producer.

```
System LQN : Producer ←Worker(rate=2.0), Consumer ←Worker(rate
    =2.0) {
  Instance qn of UL(n=2)
  true guards sync{Producer.⟨job⟩, qn.⟨enq⟩}
  true guards sync{Consumer.⟨job⟩, qn.⟨deq⟩}
  Probability ConsumerWaiting = Transient(Consumer.waiting, 10)
  Probability ProducerWaiting = Transient(Producer.waiting, 10)
}
```

Figure 5.9(a) shows the instance tree of the system with depth 0 whereas Figure 5.9(b) depicts the case that the depth is set to 1. The measures of interest for this model are the probabilities of the producer to wait for the queue to enqueue a job, and the probability of the consumer to wait for a job from the queue. As shown in Figures 5.10(a),5.10(b) and 5.10(c), the recursion depth has an impact on the equilibrium. Due to the increased redundancy of inner-queues and processing units, the overall performance increases, leading to a lower probability in the long run for the consumer to wait for a job.



(a) LQN(0): $P_{ConsumerWaiting}(t=\infty) = P_{ProducerWaiting}(t=\infty) = 0,390170$

(b) LQN(1): $P_{ConsumerWaiting}(t=\infty) = P_{ProducerWaiting}(t=\infty) \approx 0,15848$

(c) LQN(2): $P_{ConsumerWaiting}(t=\infty) = P_{ProducerWaiting}(t=\infty) \approx 0,0648$

Figure 5.10: Analysis results of the LQN models (generated by the LARES toolset)

5.4 Case Studies as Benchmarks to Testify the Scalability

The provided case-studies are used to determine how the LARES transformation scales taking different input models into account. Important measures to be taken relate to aspects of the source model, the representation of the target model and the constructed MTBDD. These metrics will give a better understanding of how model alteration may impact the generation process, the target model and the analysis of a model in order to gain insight into how the applicability of an analysis is influenced. The raised metrics can be used to assess the transformation and by doing so to determine bottlenecks in order to make improvements regarding state space or transition encodings.

Beside the transformation's efficiency, the effectiveness of a model to provide the level of abstraction in order to facilitate the analysis of the transformed model can be testified. Apart from the state space explosion problem, the increasing number of encoded transitions, in particular as a result of additional complexity due to generative/reactive expressions along the hierarchy, are considered. To give an example of model alteration, a slight change described in Section 5.3 is applied to the LQN model by substituting the <deq> forward statement in the body of the UL Module definition with

```
forward ⟨deq⟩ to maxsync{u1.⟨deq⟩, u2.⟨deq⟩}
```

This leads to a less good-natured model regarding the number of action encodings arising from the maxsync operator combined with the successive multiplication corresponding to the depth of recursion.

Table 5.1 shows the kinds of metrics that have been determined during the analysis process. The subsequent explanation is also applicable to the other tables provided in this section.

- The first part shows the metrics related to the LARES model description, comprising the lines-of-code (loc), the number of guards statements (guards), the number of Condition statements (Conditions) and the number of forward statements (forwards).
- The second part extracts the number of instantiated sequential processes, the lines-of-code of the generated SPA model and the number of action-prefixes constructed.
- The third part is related to the symbolic MTBDD encoding of the state space of the process algebra tool CASPA:
  - The number of individual action encodings are listed.
  - In order to obtain a fully *interleaved model* specification, all synchronisation sets of the generated SPA model are emptied. This allows determining the encoded (potential) state space from the resulting MTBDD and, by performing reachability analysis thereon, the (reachable) product space (such that the potential state space arising from the product of all states of instantiated behaviours

is restricted to the states reachability from the initial states of each sequential process instance).

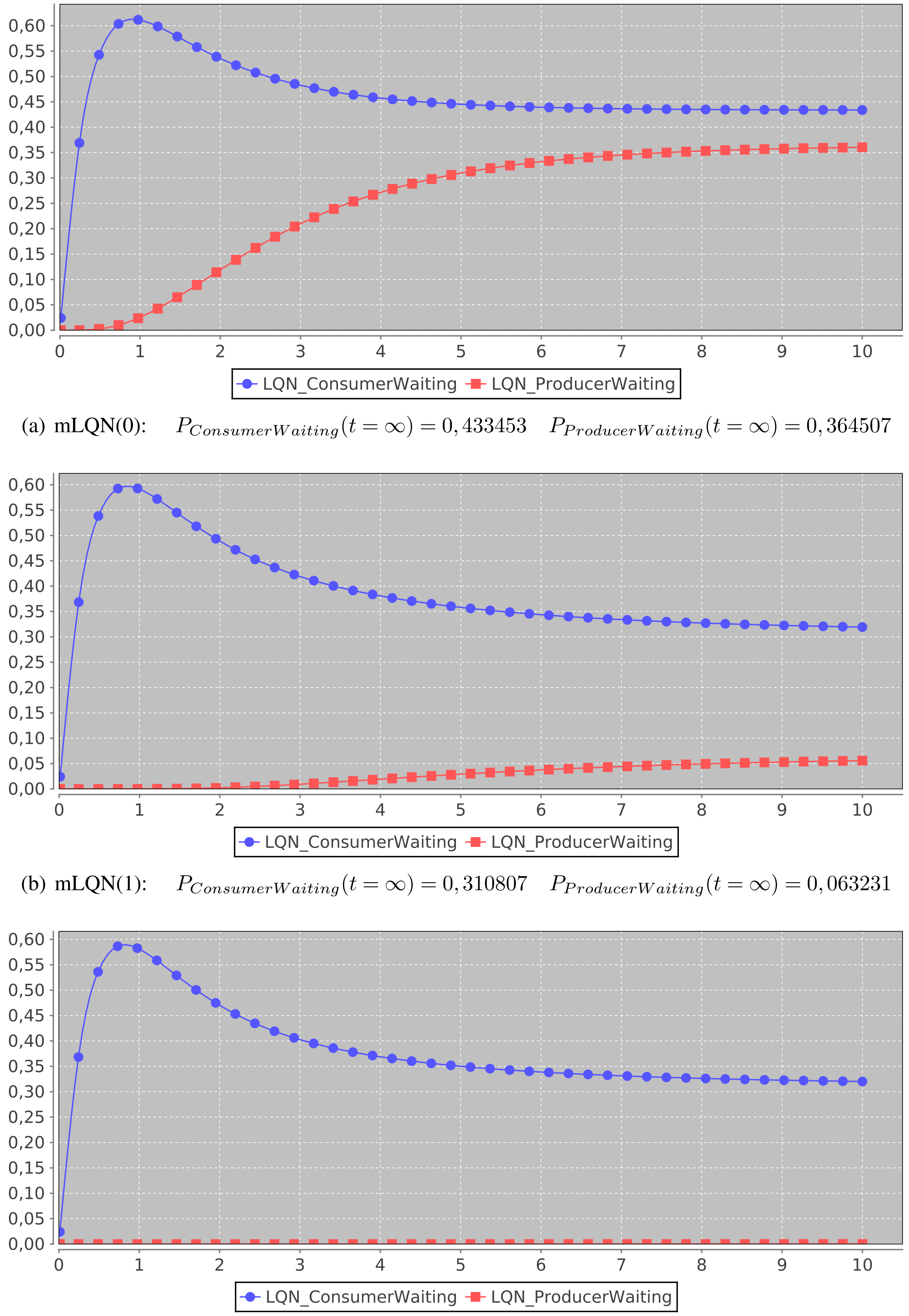
- The generated SPA model is then used to construct the *non-eliminated* MTBDD on the basis of which the reachability analysis was performed to obtain the number of reachable states, the number of MTBDD nodes, leaves (each representing a weight or rate value) and minterms (encoding the Markovian and immediate transitions).
- Subsequently, the *eliminated* MTBDD is constructed and reachability analysis performed upon to obtain the labelled Markovian transition system. Thereof, the number of reachable tangible states, the number of MTBDD nodes, leaves and minterms will be reported.

The metrics table does not include transformation duration times of the models since the analysis process always proves to be the bottleneck. It also does not state anything about the analysis durations, as this is a topic that is related to the CASPA tool and is thus not in the focus of interest. However, except for the RailCab model and the LQN model (with a recursive depth larger than two), all of these models could be analysed within seconds. As expected, the metrics of the assumed ”not that good-natured” modified models exhibit an increased number of action label encodings within the MTBDD in comparison to the original models. The higher the recursion depth, the higher is the multiplicative effect of the recursion and the arising combinatorics from the maxsync operator of the modified LQN versions (`mLQN(n)`). The high number of action prefixes is, apart from the large number of action labels, owed to the many process instances involved in an interaction.

The reason for the equality of the size of the encoded space, the product space and the reachable states before elimination is, in contrast to the examples later dealt with, given by the fact that the queue behaviour has 2 states, the processing behaviour has 4 states and the worker behaviour has 2 states. All of them follow the n’th power of two and can thus be encoded by a minimal number of Boolean variables. Furthermore, each state of a process instance is reachable from the provided initial state. To give an example, the size of the binary encoded space for LQN(0) can be calculated by

$$\overbrace{(\underbrace{2^1}_{\text{worker encoding}})^2}^{\texttt{Producer}\text{ and }\texttt{Consumer}\text{ instances}} * \overbrace{(\underbrace{2^1}_{\text{queue encoding}} * \underbrace{2^2}_{\text{processor encoding}})^2}^{\text{two unit instances }\texttt{u1},\,\texttt{u2}} = 256$$

and corresponds (due to the previously given reasons for the LQN model type) to the formula to calculate the size of the product space, e.g. $2^2 \cdot (2 \cdot 4)^2 = 256$. Due to its recursive nature the number of process instances in the model grows exponentially. A linear growth of process instances would already lead to an exponential blow-up in the



(a) mLQN(0): $P_{ConsumerWaiting}(t = \infty) = 0,433453$ $P_{ProducerWaiting}(t = \infty) = 0,364507$

(b) mLQN(1): $P_{ConsumerWaiting}(t = \infty) = 0,310807$ $P_{ProducerWaiting}(t = \infty) = 0,063231$

(c) mLQN(2): $P_{ConsumerWaiting}(t = \infty) = 0,313650$ $P_{ProducerWaiting}(t = \infty) = 0,000229$

Figure 5.11: Analysis results of the modified LQN models (generated by the LARES toolset)

			LQN(0)	mLQN(0)	LQN(1)	mLQN(1)	LQN(2)	mLQN(2)
LARES model		loc	59	59	59	59	59	59
		guards	3	3	3	3	3	3
		Conditions	0	0	0	0	0	0
		forwards	4	4	4	4	4	4
SPA model		seq. processes	$2+4$	$2+4$	$2+8$	$2+8$	$2+16$	$2+16$
		loc of spa	85	94	155	278	295	4606
		action prefixes	12	21	24	147	48	4359
MTBDD		enc. actions	11	12	17	28	29	276
	interl.	encoded space	256	256	16384	16384	67108864	67108864
		product space	256	256	16384	16384	67108864	67108864
	bef. elim.	reach. states	256	256	16384	16384	67108864	67108864
		nodes	227	246	558	770	1538	6790
		leaves	5	5	5	5	5	5
		minterms	896	888	98304	95712	$\approx 7.38 \cdot 10^8$	$\approx 7.01 \cdot 10^8$
	aft. elim.	reach. states	87	87	3123	3123	6162243	6162243
		nodes	445	444	1699	1583	6836	5072
		leaves	5	5	8	8	16	16
		minterms	335	331	22213	21245	$\approx 8.52 \cdot 10^7$	$\approx 7.74 \cdot 10^7$

Table 5.1: Scalability metrics for the LQN model and its modified variant

size of the state space and thus explains why the number of states will increase by orders of magnitudes when increasing the recursion depth. As a consequence, only LQN models up to a recursion depth of 2 can be analysed. For the sake of completeness the analysis for the modified LQN models is provided in Figure 5.11. The system can be regarded as a construction process due to the applied modification: Whether a job can be dequeued from a unit layer depends on the availability of a job from the inner units (i.e. if there are two jobs dequeueable from the inner units, they will be combined. If there is only one single job available, it will be forwarded without being combined with another one instead). The results of the measures of interest lead to probabilities which, over time, completely differ from those of the original model as they do not converge to the same value. Its explanation can be found in the fact that, depending on the recursion depth, larger combinations of jobs may be built. For the mLQN(0) model no more than binary combinations can be build. The average waiting time for the consumer increases only slightly compared to producer's waiting time as depicted in Figure 5.11(a). For recursion depth of 1 or 2 a fourfold or even an eightfold combination may be built respectively. Thus, the gap between the curves representing the waiting probability further increases to the disadvantage of the consumer (cf. Figure 5.11(b) and 5.11(c)).

Now, the analysis of the FTN example as introduced in Chapter 1 is performed. The number of links was varied from 3 to 6 to show the combinatorics induced by guards statements and their effect on the MTBDD encoding. The increase of the number of action encodings for larger number of links (cf. Table 5.2) is less than what could be observed for the LQN model with increasing recursion depth. The reason is that in the FTN model there is no extra contribution due to the multiplicative effect of the recursion depth. Instead, the additional complexity of generative/reactive expressions leads to an increase, just not in the same order of magnitude as in the LQN setting. As implicated by the previously presented results, modellers can commonly be recommended to prevent unnecessary combinatorics as arising from the statement $\mathtt{X.x} \,\&\, \mathtt{Y.y}$ guards maxsync$\{\mathtt{X}.\langle\mathtt{m}\rangle, \mathtt{Y}.\langle\mathtt{n}\rangle\}$ within which $\mathtt{X}$ has a state $\mathtt{x}$ as well as a guarded transition $\langle\mathtt{m}\rangle$ and $\mathtt{Y}$ has a state $\mathtt{y}$ as well as a guarded transition $\langle\mathtt{n}\rangle$. The maxsync operator leads to three cases (combined with the fact that both states $\mathtt{x}$ and $\mathtt{y}$ have to be current states). Two of the cases, i.e. $\mathtt{X}.\langle\mathtt{m}\rangle \wedge !\mathtt{Y}.\langle\mathtt{n}\rangle$ and $!\mathtt{X}.\langle\mathtt{m}\rangle \wedge \mathtt{Y}.\langle\mathtt{n}\rangle$ will never be performed. In a recursive setting this might even become worse. To prevent this effect, the maxsync can safely be substituted by a sync without changing the semantics of the model, since $\mathtt{x}$ and $\mathtt{y}$ are the only states where $\langle\mathtt{m}\rangle$ or $\langle\mathtt{n}\rangle$ are possible, respectively. Another aspect which can be observed in Table 5.2 is that the encoded space, the product space and the reachable states are not equal. The reason is that the number of states s, which does not correspond to the power of two, leads to a binary encoding given by $\lceil log_2(s) \rceil$ digits by which the encodable space is not fully exhausted (i.e. $2^{\lceil log_2(s) \rceil} - s$). Moreover, the distinction of the primary link and the redundant links is realised by different initialisations of the process instances. As a result, the primary link has a reachable set of 2 states, while a redundant link has 3 reachable states. The number of states defined for the other processes (2 states for the processor behaviour, 3 states for the link repair process and 3 states for the system behaviour) are all reachable from their initial states. As a consequence, the encoded space for the FTN(3) model is calculated by

$$\overbrace{\underbrace{(2^1)^1}_{\text{primary link}} \cdot \underbrace{(2^2)^2}_{\text{redundant links}} \cdot \underbrace{(2^2)^1}_{\text{repair man}}}^{\text{network with 3 links}} \cdot \overbrace{(2^1)^2}^{\text{two processors}} \cdot \overbrace{(2^2)^1}^{\text{system behaviour}} = 2048$$

The product space is a subset of the encoded space. Its size is calculated by the formula $2^1 \cdot 3^2 \cdot 3^1 \cdot 2^2 \cdot 3^1 = 648$. It is interesting to note that in contrast to the size of the encoded space, which increases fourfold by each additional link, the set of reachable states only approximately doubles at each step leading to a moderate growth regarding the MTBDD metrics.

For the sake of completeness, the metrics of the remaining models are shown in Table 5.3. This includes the larger case study project RailCab (cf. Section 5.2) as well as the PMS model (revisited, improved and analysed within Section 5.1), and finally, the component

			FTN(n=2)	FTN(n=3)	FTN(n=4)	FTN(n=5)	FTN(n=6)
LARES model		loc	83	83	83	83	83
		guards	5	5	5	5	5
		Conditions	5	5	5	5	5
		forwards	2	2	2	2	2
SPA model		seq. processes	$4+2$	$4+3$	$4+4$	$4+5$	$4+6$
		loc of spa	118	164	234	355	587
		action prefixes	41	73	129	236	454
MTBDD		enc. actions	16	21	28	39	58
	interl.	encoded space	512	2048	8192	32768	131072
		product space	216	648	1944	5832	17496
	bef. elim.	reach. states	132	300	636	1308	2652
		nodes	362	576	852	1336	2051
		leaves	7	7	7	7	7
		minterms	460	1232	2944	6704	14896
	aft. elim.	reach. states	32	64	128	256	512
		nodes	326	511	707	909	1360
		leaves	6	6	6	6	6
		minterms	87	203	467	1059	2371

Table 5.2: Metrics for the FTN models with increasing number of Networks n

monitoring system model from Section 1.3. To complete interpreting the analysis results obtained for the PMS model, the size of the encoded space can be validated by a formula, where again, the odd number of reachable states for the system behaviour and for switch s[5] (which has 3 potentially reachable states due to its initial setting, instead of the 4 states of the other switches) lead to an encoded space of size

$$\overbrace{(2^3)^1}^{\text{system behaviour}} \cdot \overbrace{(2^2)^4}^{\text{switches 1-4}} \cdot \overbrace{(2^2)^1}^{\text{switch 5}} \cdot \overbrace{(2^1)^2}^{\text{components}} = 32768$$

which is a superset of the product space calculated by

$$\overbrace{(5)^1}^{\text{system behaviour}} \cdot \overbrace{(4)^4}^{\text{switches 1-4}} \cdot \overbrace{(3)^1}^{\text{switch 5}} \cdot \overbrace{(2)^2}^{\text{components}} = 15360$$

Due to the high dependency in the PMS model, a heavy reduction in the size of the reachable state space can be observed in contrast to the fully interleaved model.

To subsume, considering the formulas and the intuition generated by a number of examples, modellers should be able to give an estimate how large a model might be, and whether

			RailCab	PMS	CMS
LARES model		loc	330	82	101
		guards	12	5	5
		Conditions	23	4	3
		forwards	14	2	5
SPA model		seq. processes	23	8	6
		loc of spa	46796	283	506
		action prefixes	46365	176	417
MTBDD		enc. actions	3233	31	73
	interl.	encoded space	134217728	32768	2048
		product space	100663296	15360	1152
	bef. elim.	reach. states	70219776	2511	88
		nodes	36196	2047	672
		leaves	10	14	6
		minterms	$\approx 1.01926 \cdot 10^9$	9035	303
	aft. elim.	reach. states	3188736	915	31
		nodes	6539	1499	531
		leaves	9	14	8
		minterms	$\approx 3.98715 \cdot 10^7$	2131	95

Table 5.3: Metrics for the RailCab, the Phased Mission System and the Component Monitoring System models

the analysis will be possible using e.g. the CASPA tool. The state space can be over-approximated as shown for the encoded space or even further narrowed by the product space. The number of reachable states may be estimated by the order of magnitude of inter-dependability, expressed in terms of synchronisation. Furthermore, the number of transition encodings can be calculated with regard to the complexity of generative/reactive expressions and the combinatorics arising from the hierarchy. Nevertheless, the intrinsic complexity of a system can only be reduced by abstraction. It is left to a modeller, which level of abstraction to choose. Still, knowledge about the transformation process might help modellers to prevent specifying non-intrinsic and thus unnecessary state and transition combinatorics.

Chapter 6

Approaches Related to LARES

A lot of approaches can be enumerated that devote themselves to dependability analysis. An almost exhaustive overview is given in [99]. Many of them may be considered as higher-level dependability languages which build on top of other formal languages such as Markov chains. For the purpose of generating a model in the target notation and to make use of the specifically developed tools, numerous mappings have been defined in terms of transformation rules.

In order to distinguish these higher-level approaches, three categories will be introduced, each consisting of several criteria for comparison and classification of the considered approaches:

1. **Language Aspects**

   (a) Expressiveness: Is the language of general purpose or does it only provide specific constructs to model some restricted aspects of a domain?

   (b) Area of deployment: Is the language suited as a stand-alone modelling language or may it only serve as an intermediate formalism?

   (c) Structuredness/Modularity: Does the language provide constructs to reflect a system's structure (flat/planar vs. hierarchical/structured) and the ability to substitute parts of a system easily (monolithic vs. modular)? How are the component entities composed and how do they interact?

   (d) Level of abstraction: Does the language provide high-level language constructs?

   (e) Separation of concerns: Does the formalism support distinguishing between structural description, sequential behaviour (nominal, faulty) and interaction behaviour?

2. **Transformation Aspects**
   (a) Target Formalisms: Which types of target formalisms are addressed as semantic domains and do they capture the whole expressiveness of the source language?
   (b) Formality: How are the implemented transformations described (formally, by example or by code)?
   (c) Automation: Is the analysis fully automated or is manual intervention required?
3. **Analysis Aspects**
   (a) Type of analysis: Analytic vs. statistical (e.g. simulation) approaches to obtain dependability / performability measures, model checking of temporal properties, optimisation approaches, ... ?
   (b) Integration: Is the solver integrated such that users are spared from dealing with external tools?

Some previously enumerated criteria are ingredients of the concept of an architecture description as defined in the ISO/IEC/IEEE 42010 standard [88]. An architecture is required to capture the "aspects considered fundamental about that system in the context of its environment". Commonly, an architecture encompasses the system properties embodied in its elements, is able to represent the system's structure, consisting of instantiated substructures, their relationships and evolution over time. An architecture description language should provide the ability to create, refine, validate and analyse the architecture, so that the "[...] concerns of the diverse stakeholders can be addressed [...]".

The authors of [98] concluded that "[...] 1) while practitioners are usually satisfied with the design capabilities provided by the languages they use, they are dissatisfied with the architectural language analysis features and their abilities to define extra-functional properties; 2) architectural languages used in practice mostly originate from industrial development instead of from academic research; 3) more formality and better usability are required of an architectural language", which encourages providing "extra-functional properties", sound semantics and better facilities to describe and analyse the systems of interest.

Beyond "extra-functional properties" such as pure dependability aspects, a language for dependability analysis is required to capture also *dynamic reconfiguration*. Following [86], reconfigurations can be classified into the aspects *implementation* (i.e. when modules are replaced by others, maybe due to providing a more efficient implementation), *structural* (i.e. when the logical structure and interconnection of modules change) and *geometric* (i.e. mapping of the logical structure to the underlying platform, for example a distributed architecture). Furthermore, events that cause a reconfiguration are distinguished by the author as being synchronous (corresponding to the current process state/step) or asynchronous (independent of the current process state/step), both in general entailing non-negligible costs.

This chapter focuses on a set of selected approaches that fall into the category of higher-level languages which are based on other low-level formalisms. Some of them are already surveyed in [99] next to numerous low-level formalisms, while others appeared later or haven't been considered there. The chosen approaches are examined following the criteria for the different categories, and how well they perform when it comes to capture the dynamics of reconfiguration. The comparison shown here does by no means claim to be complete in the sense that there probably exist further formalisms in this particular domain.

Indeed, many formalisms are used for dependability modelling that are considered too low-level or weak with respect to the power of the language. Thus, Petri net (PN) or Markov chain (MC) approaches are skipped, as well as standard Reliability Block Diagram (RBD) or Fault-Tree (FT) approaches. Nevertheless, the following listing will serve as an overview of more sophisticated languages, it will give a clue of the characteristics of these approaches and it will help emphasise the requirements and the resulting incentive to develop LARES:

- Arcade is a language for modelling and evaluation of dependable systems [22]. It provides language means corresponding to templates representing the building blocks of a system.
- OpenSesame (Simple but Extensive Structured Availability Modeling Environment) has increased modelling power with respect to FT or RBD [139] by offering specific diagrams and constructs (e.g. in order to model common cause failures, failure propagation and non-perfect failure detection). The specifications are translated to stochastic PNs and solved by the tool DSPNexpress.
- In [26], a modelling language based on dynamic fault tree (DFT), as an extension to fault trees, is presented formally by means of I/O-IMCs in order to resolve limitations in modular analysis and model building. A similar approach has been introduced in [91], where the formal semantics of a DFT has instead been defined by means of the process algebra PEPA.
- The Failure Automaton (FA) as given by [111] serves as a simple intermediate language which is specified using the Z language in order to provide a well-defined semantics (based on predicate logic and set theory). On top of the FA a number of high-level reliability formalisms have been defined to emphasise the benefit of this methodology.

- KLAPER (Kernel LAnguage for PErformance and Reliability analysis) has been presented in [74, 76]. There, a model-driven approach is proposed that defines an intermediate language that, dependent on a parametric transformation specification, transforms into a refined model which afterwards may be converted to a specific solver input model.
- MoDeST (Modeling and Description Language for Stochastic Timed systems) [54] is a process-oriented language for the analysis of dependable systems and resembles structural programming languages at the level of sequential behaviour definitions.
- SLIM (System-Level Integrated Modeling) [35] is a model-based hierarchical language which was developed within the COMPASS project. It is aimed at detecting flaws in early design-stages of hardware/software systems in an automated manner.
- AltaRica [114, 115] is aimed at modelling discrete systems and provides means to reflect a hierarchical system structure. It is mainly used for functional verification.

The approaches from [26] and [91] are excluded as DFTs are known to be too restrictive. As specific dependability constructs lead to restricted expressiveness, OpenSesame [139] is omitted too. The remaining approaches are detailed in the following sections and examined by the criteria posed before. Firstly, ARCADE as a representative of a user-friendly template language is detailed in Section 6.1. Secondly, FAs are dealt with in Section 6.2, since the formalisation of the language syntax and semantics using the Z language is very interesting. In Section 6.3 KLAPER as a model-driven approach is discussed, followed by Section 6.4 presenting MoDeST which resembles imperative structured programming. Section 6.5 considers SLIM which seems to be the approach related most closely to LARES and finally, Section 6.6 details the relation to the wide-spread and well-known formalism AltaRica. The chapter is concluded with Section 6.7 by exposing all approaches to the criteria and emphasising their advantages and deficiencies, which helped to elaborate the desired characteristics and the incentive of developing LARES.

6.1 Arcade: Architectural Dependability Evaluation

The Arcade toolchain is used for dependability evaluation (see [22, 23, 24]) and offers different input languages such as the specific Arcade formats. As a final goal, UML [108] and AADL [59] should be addressed as potential input formalisms. The underlying semantics is given in terms of Input/Output-Interactive Markov Chains (I/O-IMC) as defined in [25], i.e. an automaton composed of states and discrete (input, output and internal) actions used to define interactive transitions among the states.

Three main building blocks which interact via input/output actions are defined for Arcade: A basic component, a repair unit and a spare management unit.

> **Basic Component:** The basic component is a building block which is used to specify physical/logical system components. It comprises predefined operational modes organised in groups (e.g. active/inactive, on/off) including the transitions between them. The modes within a single group are mutually exclusive. Further, a failure model can be attached to each operational group specifying which kind of failure can occur. The operational modes and the failure model are then superimposed such that an I/O-IMC can be derived.
>
> **Repair Unit:** A repair unit may apply one of the predefined repair strategies, i.e. *dedicated repair* and variants of *first come first served*, which may include preemptive or non-preemtive priorities, defined in terms of I/O-IMCs comprising failure signals as inputs and repair signals as outputs. The authors of [24] stated that some repair strategies could lead to large state spaces depending on the number of components. Unfortunately, it is not mentioned whether these strategies can be generated automatically for an arbitrary number of components.
>
> **Spare Management Unit:** Two configurations are predefined in terms of I/O-IMCs for the spare management unit, i.e. *one primary and one spare* and *one primary and two or more spares*.

The syntax of Arcade resembles different templates for each building block which is to be completed by the required aspects a building block might capture. To give an example: For the repair unit a name, the involved components, a predefined strategy and depending on the chosen strategy a priority for each component have to be specified. The language can be extended by additional constructs defined by means of an I/O-IMC and the required syntactical elements. The authors admit in [24, Section 4] that the translation process of the building blocks has not been automated yet. All resulting I/O-IMCs are subsequently composed automatically, interleaved by aggregation steps (which implies the order of composition to be predetermined by the users to avoid running into the state space problem during composition). The final I/O-IMC is redirected to the CADP tool [62] which transforms the I/O-IMC to a CTMC and performs the analysis (cf. Figure 6.1).

6.2 FA: Failure Automaton

Most approaches defining languages for reliability analysis have imprecise semantics: They have a well-defined syntax but lack a complete semantic definition because their semantics is often only sketched for a number of examples. As a consequence, the authors

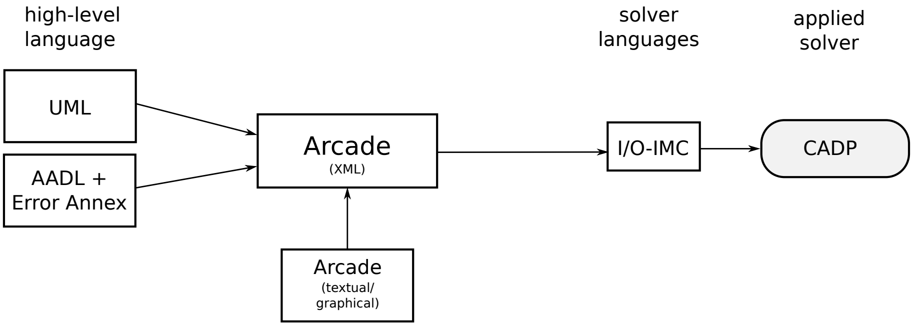


Figure 6.1: Arcade: High-level language mappings (compiled from Figure 1 in [22])

of [111] claim that an intermediate language as a mathematical precise abstraction should support higher-level languages. By providing a common formal semantic domain, higher-level languages with precise meaning can be developed and implemented at reduced cost. Their proposition thus is to use a formally defined intermediate reliability modelling language instead of poorly understood modelling formalisms that may be semantically ambiguous. They claim that an intermediate language may help reduce the design and implementation cost due to the reduced *distance* between a high-level language (such as RBD, DFT and Boolean logic Driven Markov Processes (BDMP) [29]) and the low-level target formalism as illustrated by Figure 6.2. Furthermore, the general confidence in consistency, in the meaning and the implementation is higher because of an existing well-defined formal specification. Their intermediate language specification Failure Automaton (FA) is denoted using the Z language [132]. A FA is a state-based language similar to a Markov model. A state comprises the state of events (*failure status* or *activity status*), a history (failure sequence to construct high-level features depending on the order of events), spares in use (keep track of spare allocation) and system status (system failure and failure coverage). It includes features such as repair on demand (ROD). Each transition is labelled, representing a *causal basic event*. A transition either corresponds to a failure or a repair. One distinguishes between *basic events* (occurring on their own) and *derived events* (representing the interaction of events). While the derived event interaction is described by the FA, the *Basic Event Model* describes the stochastic behaviour of a basic event which is represented by four Markovian transitions (two of them represent a failure event and the other two represent a repair event which may take place in both the `active` or the `inactive` state). Further it includes a coverage model (probability of event occurrence causing a complete system failure) and a transfer model (to provide a probability feature such as ROD). The authors argue that their separation of concerns avoids having fixed rates and thus a large FA doesn't have to be completely recomputed for the case that basic event properties change.

Unfortunately the authors did not publish the complete Z specification of the FA language. Since FA was originally developed to capture DFTs, for sure also RBDs can be easily

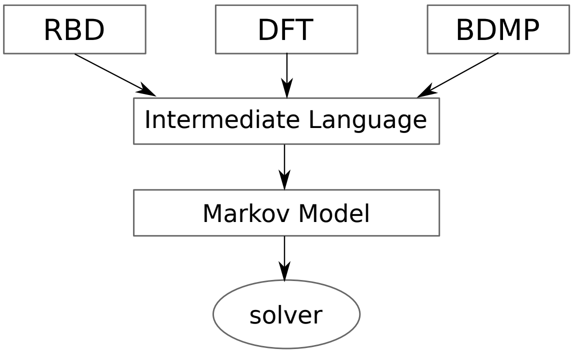


Figure 6.2: FA: High-level language mappings (compiled from Figure 1 in [111])

described, since they only represent a Boolean redundancy structure. Even by additionally defining BDMP upon their modified version of FAs, the generality of their approach as an intermediate language for reliability modelling remains unproven. All the considered high-level languages only allow Boolean models and it remains unclear whether the provided features are sufficient for all kinds of reliability considerations. Further, since FA still is a very low-level formalism it will be questionable if the *distance* is that much lower than targeting directly to other Markov models. Still, the way of using the Z language for creating well-defined semantics for a formal language would be desirable in general. The full disclosure of the FA specification could attract language developers to adopt the FA approach and to define higher-level languages resting upon.

6.3 KLAPER: Kernel LAnguage for PErformance and Reliability analysis

In [75], a language has been presented that is designed as an intermediate language, aimed at bridging the semantic gap between design-oriented and analysis-oriented notations. As the name KLAPER (KErnel LAnguage for PErformance and Reliability analysis) implies, it is designed as a compact language to capture only the relevant information of a (software) system required for performance and dependability evaluation. For KLAPER a MOF (Meta-Object-Facility) compliant meta-model has been defined [107]. Operations of a software system can be represented by *steps* which take some time delay or may fail. Steps are grouped in *behaviours* which may either be used in the context of *workload*s or *resource*s. For KLAPER, a component-based system is an assembly of interacting resources that offer *services*.

In [75], a workflow is sketched starting from an OWL-S [102] service description language which is mapped to KLAPER and subsequently transformed into an extended queueing network (EQN) that is solved using the Software Performance Engineering (SPE)

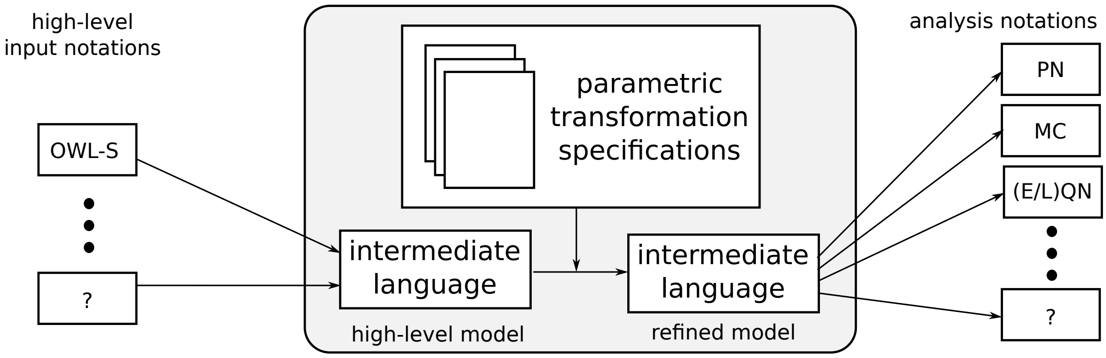


Figure 6.3: KLAPER: Workflow of language mappings (compiled from Figures in [75, 76, 77])

framework [129]. It seems as if KLAPER was defined having service descriptions in mind, since many elements of OWL-S nearly directly correspond to elements provided by KLAPER. Anyway, since the standard OWL-S notation does not provide QoS (Quality of Service) attributes, some KLAPER elements remain unspecified after transformation to the KLAPER language and need to be completed to carry out the analysis. In [77] the KLAPER-D extension has been developed to capture the dynamic reconfiguration behaviour of component based systems. It starts again from an OWL-S service description that is mapped to KLAPER-D and then transformed into a *semi-Markov reward* (SMR) model which can be analysed by the SHARPE tool [85]. In addition to the OWL-S design notation in [74] the UML profile for Schedulability, Performance and Time (UML-SPT) [106] is considered an additional input formalism. Furthermore, a target notation mapping to layered queueing nets (LQN) [60] is presented and illustrated by a case-study that models some use-cases of the CoCoME example. All in all, the mapping workflow between the design-oriented input formalism and the output analysis notations are summarised by Figure 6.3 following [74, 76]. It is worth mentioning that the KLAPER approach makes extensive use of Model Driven Development (MDD) techniques. Meta-models are defined using the MOF standards and the QVT language is used to represent the transformation rules [107]. The paper states that the whole process is not fully automatic. A number of tools have been developed around KLAPER for the Eclipse platform that encompass a meta-model plugin, an editor plugin, the LQN meta-model plugin, an LQN editor plugin and the corresponding transformation plugin. However, in [74, p. 329] the authors concede that KLAPER is ‘inherently based on a client-server architectural model. Other styles [...] are not considered yet’ and that the implementation is still incomplete.

Very recently, the KlaperSuite was presented [41]. It is an integrated framework built upon the KLAPER language which targets several formalisms or solvers (such as PRISM, RMC (Recursive Markov Chain), SimJava and LQN). Furthermore, it is claimed that now the transformation process is fully automated.

6.4 MoDeST: Modeling and Description Language for Stochastic Timed Systems

In [54] the MoDeST language has been introduced to model stochastic and real-time systems. It is a formally defined descendant of a process algebra language whose syntax resembles the C language enriched by constructs for data modularisation, exception handling via try/catch blocks and concepts known from real-time and stochastic discrete event systems such as probabilistic branching, clocks and random variables. Its formal semantics is given by means of stochastic timed automata (STA) [18]. Processes can be composed in parallel or manipulate typed data variables via assignments. Variables can be declared, whose specific scope is either local or global. All clock-typed variables run at the same speed. The available predefined probability distributions encompass exponential, uniform and normal distributions. Interactions among composed processes are realised by *action*s that can be guarded and whose semantics underlies a maximal progress assumption. Internal tau-actions are not attainable for synchronisation. Besides, exceptions can be raised and handled in catch blocks as known from general programming languages. Alternatives can be specified, as a number of choices between different possible behaviours which could be probabilistic, as well as repetitive behaviours.

In [83] the UML-based high-level formalism Stochastic State Chart (StoChart) is addressed to define a transformation to MoDeST and thus allow using the MOTOR/Möbius toolset [19, 46] for the analysis of StoChart models. MoDeST is too expressive to be fully captured by a single analysis tool. The *single-formalism multiple-solution* approach can thus only be realised by focussing on restricted subsets that can be handled by some specific analysis method (i.e. as implemented by specialised tools) [81]. To give an example, there is also an approach denoted as *mcpta* to perform model-checking of probabilistic timed automata specified in the MoDeST language. For that purpose, the model to be checked is transformed following a predefined mapping of a subset of the MoDeST language to PRISM's [95] guarded-command based language [82]. For this purpose, the MoDeST language has been extended by *properties* that are included in the transformation such that their corresponding PRISM properties are generated. Recently, MoDeST has also been extended to support stochastic hybrid automata [79].

The MoDeST toolset has meanwhile been enriched by numerous tools (i.e. *mcpta* to perform model-checking for the probabilistic timed automata subset, *mctau* [16] to model-check corresponding to UPPAAL's [12] networks of timed automata subset, the discrete-event simulator (DES) *modes* [17] which includes a statistical model checking (SMC) feature (and replaces the former MÖBIUS-based simulator), the graphical user interface *mime* and the visualisation tool *mosta* [81]). For an overview of the mappings and the tools that have been developed see Figure 6.4.

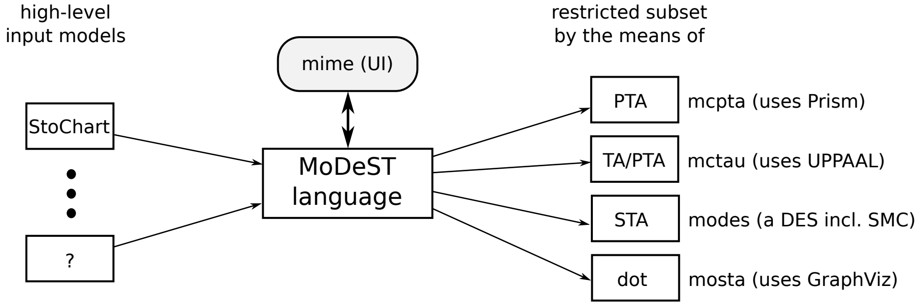


Figure 6.4: MoDeST: Workflow of language mappings

6.5 SLIM: System-Level Integrated Modeling

A number of approaches address AADL [59] as a high-level input language by underlying a formal semantics thereof. However, the most comprehensive approach results from the COMPASS (COrrectness Modeling and Performance of AeroSpace Systems) project that aims to develop “[...] a coherent and multidisciplinary approach towards developing systems at architecture [...] level [...]” [31] and finally provides a toolset to analyse AADL models including the Error Model Annex [32]. It was desired to specify the nominal and the faulty behaviour of a system, to describe sporadic and permanent faults (for hardware and software), error propagation and degraded modes of operation. Furthermore, the notions of observability as well as timed, hybrid and probabilistic behaviour were required to be supported. In order to capture all these aspects, the SLIM (System-Level Integrated Modelling) language [35] has been defined “[...] in order to provide a cohesive and uniform approach to model heterogeneous systems, consisting of software [...] and hardware [...] components, and their interactions” [36].

SLIM supports system hierarchies, is component-oriented and distinguishes between software (e.g process, thread, thread group) and hardware (processor, memory, device, bus) and composite components (providing specific language constructs for each type). Hereby hierarchical structures can be specified. Components are abstract and define a type providing a specific interface. Each component may be implemented according to the interface. Continuous variables can be defined in the implementation for capturing continuous aspects of a system. The language provides constructs to define event and data ports for inter-component communication. Ports are used to interconnect sibling components and subcomponents and may depend on the current mode such as clock invariants, flow expressions (which affect continuous variables) as specified in the mode transition system. The nominal behaviour will thus be described by mode transitions which act on the occurrence of a triggering event in case that a guard is satisfied and may perform an assignment.

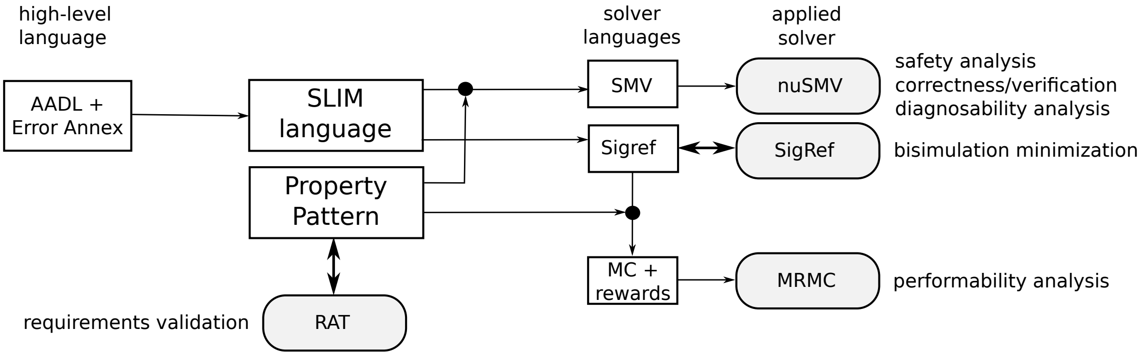


Figure 6.5: SLIM: Workflow of language mappings and tools

An error behaviour can be specified as a companion behaviour to the nominal behaviour. Error states and incoming or outgoing error propagations can be defined as interface, whereas its implementation consists of transitions that depend on the defined events which lead to the actual error state. The nominal behaviour needs to be modified to integrate the error extension by adapting nominal transitions, adding fault injection transitions, nominal transitions with fault effects and error propagation transitions.

As a formal basis for the SLIM language, the concept of a Network of Event-Data Automata (NEDA) has been introduced (which itself is defined by means of state-based transition systems which require continuous variables to be discretised) in order to give SLIM a formal semantics [35]. Since the generated state space of a SLIM model could be large, especially due to continuous variables, Event-Data Automata are symbolically encoded in terms of Boolean formulas, which hence generalises as well to a NEDA.

Numerous analysis methods dealing with functional correctness, safety and dependability, performability, fault detection, fault identification, fault recovery effectiveness or requirements validation can be performed with the help of the developed toolset as shown in Figure 6.5, derived from [33]. The COMPASS toolset employs a number of mature tools: For that purpose, the transformation to the SMV notation allows the NuSMV (real-time, probabilistic) model checker [43] to be applied. Therefore, the requirements have to be posed as CTL, LTL formulas or by the Property Specification Language (PTL), which extends LTL by the power of regular expressions. Advanced abstraction techniques have been integrated into nuSMV to avoid the state space explosion problem which especially arises due to the discretisation of continuous variables, e.g. performing abstraction refinement guided by counterexamples [47] and in addition applying Satisfiability Modulo Theory based constraint solving [40, 44]. In order to calculate whether specific performance requirements are met, the MRMC model checker [90] can be invoked.

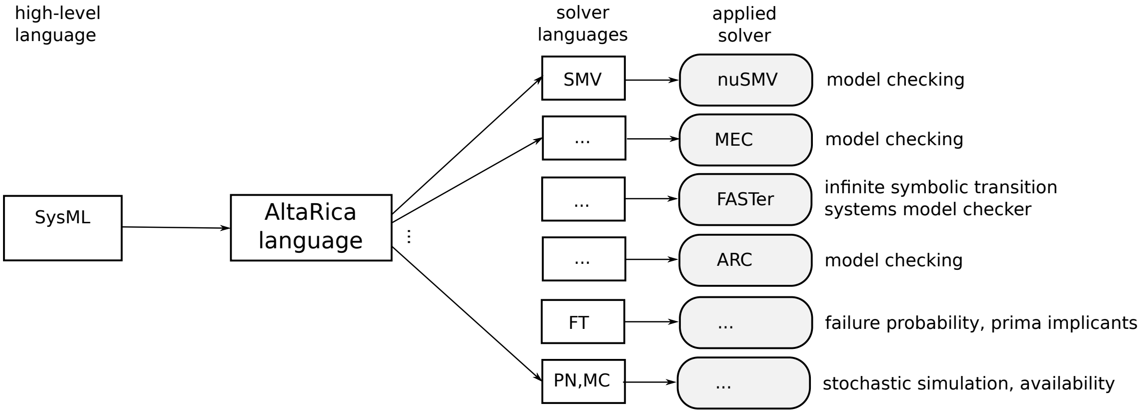


Figure 6.6: AltaRica: Workflow of language mappings and tools

6.6 AltaRica

AltaRica is a mature language that was first presented in [114, 115] and has continuously been maintained and developed since then. It is designed to model functional and non-functional behaviour of critical systems to perform dependability analysis. Similar to other high-level approaches, the aim was to reduce the distance from a modeller's perspective to rather low-level models, thereby facilitating the design of complex systems. AltaRica provides a textual as well as a graphical representation and compiles to low-level formalisms such as fault trees, Petri nets or Markov chains. The semantics of AltaRica is given in terms of constraint automata (CA) [37].

The DataFlow subset of AltaRica is based on mode automata (MA) [117] and is claimed to be easier to compile. In this subset, states are implicitly defined by state variables and a transfer function (which depends on the current state and the input variables) controls the output variables. Therefore, events can be defined which alter the state variables. Those mode automata can be composed by denoting their interconnection and specifying the synchronisation of events, thus allowing a hierarchical system structures to be defined. Events may be timed, comprising a probability distribution (e.g. exponential, Weibull), or instantaneous (i.e. distinguishing immediate or conditional events that may be prioritised). In [39] a *timed extension* was introduced for AltaRica, thus allowing real-time aspects to be captured. And as described in [3], flow-variables were added to incorporate continuous aspects of "multi-physical systems". Besides, several tools are implemented to support the analysis. As only partly surveyed by Figure 6.6, mainly model-checking techniques are implemented by the tools ARC, MEC [8], FASTer, NuSMV [34] and others. But other analysis techniques are addressed as well, such as compilers for fault-trees [63] and Markov Graphs, stochastic simulators, minimal cut set or sequence generators, or compilers to other formal languages such as Lustre and SMV, which in turn allow their tool implementations to be applied.

Since AltaRica as a high-level formalism decreases its distance to other architectural languages such as SysML, it can also serve as an intermediate formalism as shown in [49]. AltaRica enjoys great support from industry partners and can thus rely also of a number of complex case-studies, which in turn contribute to the whole AltaRica workbench and development of the language.

6.7 Classification and Distinction

The previous sections gives an overview of a number of selected approaches, dealing with language definitions, their semantics, the analysis methods and developed toolsets. This section aims to provide a survey of the important characteristics and differences among these approaches and to LARES. Table 6.1 attempts to depict the relevant characteristics to compare the selected approaches. The table contrasts the ability of a language to capture a broad range of different models by its expressiveness, the manner and purpose of its deployment (considering the case that an end-user is addressed or the language serves as an intermediate representation), the ability to achieve modularity of the specified component types (in order to allow successive embeddings as key features to produce structured and reusable descriptions to facilitate compositional modelling), the description of dependencies among the components, its level of abstraction (which often corresponds to the modelling effort of a language) and whether the different concerns can be separately described. Furthermore, the transformations into the target notations are addressed by discussing whether they can reflect the whole expressiveness of the language or the case that a number of submodels are addressed. The table contrasts also the level of formality of the description of the semantics and the degree of automation. And, finally, it reflects which kinds of analysis methods can be applied and whether the analysis workflow integrates external solvers smoothly.

From a modeller's perspective, the idea to design Arcade as a template-based language is beneficial if the desired aspects conform to the expressiveness of the standard Arcade language. Otherwise, the formalism has to be extended, which requires modellers to deal with the fairly low-level I/O-IMC formalism. Complex reconfiguration aspects might be difficult to incorporate into the language as well as specific error behaviour or repair behaviour that goes beyond the predefined constructs. Furthermore, the distance to other high-level languages such as AADL seems to be quite large, as Arcade does not even support for reflecting a system's hierarchy. Another issue is that the transformation process of building blocks is not fully automated, which contradicts a neat workflow. We pursued a similar approach of composing predefined patterns for the ZuVerSicht formalism [15]: Similarly to Arcade, this formalism could be extended, but this requires doing implementation work by defining new process algebra patterns. Predefined patterns in

		Arcade	FA	KLAPER	MoDeST	SLIM	AltaRica	LARES
language	expressiveness	-	o	+	++	++	++	++
	hierarchy/structure	-	-	-	+	++	++	++
	modularity	+	+	+	+	+	+	+
	level of abstraction	+	–	o	o	o	o	o
	separation of concerns	++	-	+	o	++	+	+
	purpose of deployment	i&u	i	i	i&u		i&u	i&u
semantics & transformation	semantics by means of	I/O-IMC	CTMC	EQN	STA	NEDA	CA/MA	CTMC/MDP
	target languages	I/O-IMC	CTMC	EQN	many	SMV, MC+rew.	many	SPA/LTS/MDP
	degree of automation	o	+	o	+	+	+	+
	extensibility	+	+	+	+	+	+	++
analysis/tools	analysis methods	o	o	o	++	++	++	+
	tool support	-	–	o	++	++	++	+

Table 6.1: A survey to contrast some characteristics between selected approaches, where i=intermediate, u=end-user and the rating is as follows { – ≙ 🟥, - ≙ 🟧, 0 ≙ 🟨, + ≙ 🟩, ++ ≙ 🟩 }

terms of low-level formalisms were considered unnecessary for LARES in order not to limit the expressiveness and extensibility of its language features. Instead, the language constructs LARES offers will be used to define reusable “higher-level construct libraries”.

As the FA formalism is designed to serve as an intermediate language, it is rather low-level. The focus of that approach is to provide its semantics by the Z-notation to serve as a basis to define higher-level languages. The issue with this approach is that neither the language specification seems to be available nor that there is any proof of how applicable this approach is to higher-level languages with higher expressiveness than RBD, DFT or BDMP. However, it might be very cumbersome to start from such a rather low-level formalism specification in order to base more advanced languages with constructs of high abstraction upon the FA description. For LARES, the aim was not to give a formalisation by a formal language such as the Z-language, but to provide an exhaustive formalisation by its syntax specification, its abstract representation and by specifying two distinct transformations which produce behaviourally equivalent models with respect to the notion of strong bisimulation.

KLAPER takes a purely model-driven development (MDD) approach to specify meta-models and transformations. It serves as an intermediate formalism. The KLAPER formalism is tailored to client-server architectures. It therefore suits well for web-service descriptions. However, this could limit its field of application when it comes to other kinds of systems. The application of MDD techniques for LARES was also an important design choice, but is not applied as pervasively as in the KLAPER approach. For LARES, a rather pragmatic use of MDD has been pursued in order to develop Eclipse-based tools and to serve as interface to other application level formalisms.

MoDeST is a very concise, easy and expressive language which allows applying various tools for analysis in order to evaluate several kinds of measures of interest. It is not hierarchical in the way of a design/architectural language, nevertheless hierarchical structures can be specified. The aim to serve as a design language was a crucial requirement for the definition of LARES to enable modular and hierarchical modelling.

Due to the large community and great support from industry, AltaRica got very mature over time. By now, a number of expressive extensions and a variety of analysis techniques have been developed in order to capture also other types of models. AltaRica provides its tools for download under the LGPL license, which makes it easy to integrate the approach within other projects.

Considering only expressiveness and available analysis back-ends, MoDeST, AltaRica and SLIM might be better positioned than LARES, since they also provide constructs to capture real-time and hybrid aspects or stochastic distributions that go beyond exponential ones. For this purpose, they bind to multiple solvers which directly support these concepts. LARES has focussed on extensibility and strict hierarchical modelling. With the upcoming extensions, i.e. reward extension and decision extension [69], LARES language expressiveness has extended towards rewards and non-determinism. The underlying target language is thus given by a Markov Decision Processes (MDPs) that can be used to calculate optimal policies considering the defined rewards. Nonetheless, the SLIM language has meanwhile received a great backing by the powerful predicate abstraction techniques integrated into the nuSMV model checker. These advanced techniques would serve as potential solutions to tackle with non-Boolean variables addressed by first-order constraints. These could be introduced within Behavior definitions or as an extension to condition expressions to LARES in the future as described in Chapter 7. While in LARES an error behaviour is not explicitly denoted by a keyword and modellers have to care which states to address by some measure, in SLIM models, the error model has to be connected to the nominal behaviour by a fault injection-specification. This procedure probably feels a bit odd for some modellers since they have to consider the composition of the two behaviours. Since there is no distinction between error and nominal behaviour in LARES, the interconnection is realised by standard guards statements.

Very recently, in [4] an overview of approaches about model-based design of dependable systems was given, discussing the limitations and evolution of analysis and verification approaches. LARES has been compared among other model-based dependability analysis approaches whether they overcome the limitations known from event-based approaches such as fault trees. LARES and a few other (mostly model-based transformational) approaches (including AltaRica) have been categorised to overcome all these limitations such that they offer the capability to handle temporal notions, provide constructs to capture the architectural design incorporating the required dependability aspects, perform an automatic translation into analysis notations and are thus less dependent on a modeller's experience or time invested. They can finally capture the component-wise nature of embedded systems to enable a high-level of reusability. In the category of model-based verification the SLIM language of the COMPASS project is also addressed as an integrative approach providing fault injection techniques, among other such as the FIACRE language [14] as a formal intermediate language for SysML models.

To summarise, the languages SLIM and AltaRica are closely related to LARES. Especially SLIM provides many features LARES does not yet offer, not at least because of the powerful nuSMV solver. Despite shortcomings in expressiveness and capabilities of the addressed back-ends, LARES provides reasonable constructs in order to define complex interaction patterns which arise in dependable systems (based on the current state and the possible subsequent behaviour). Possible shortcomings in the language expressiveness could easily be resolved by adapting the transformation, probably by the cost of a larger state space or a larger number of transitions. Further extensions to the language may require addressing solvers other than CASPA in order to be handled adequately.

Chapter 7

Conclusion and Future Work

This thesis provides the complete definition of LARES by covering all relevant aspects such as the language (i.e. its concrete and abstract syntax), its semantics expressed by means of two distinct transformations and how the validity of these transformations is guaranteed. As depicted in Figure 7.1, one of the transformations defines rules and functions which map a LARES model to a stochastic process algebra model as used by the performance and dependability evaluation tool CASPA. The other transformation denotes operational semantic rules to determine a weighted ESLTS, which finally leads to a labelled continuous time Markov chain after performing elimination of vanishing states. As CASPA implements the formal semantics of its SPA, which is given by means of labelled transition systems, and supports the elimination of vanishing states, the outcome is again a labelled Markov chain. Both transformations are defined in a manner such that they finally produce behaviourally equivalent models following the notion of strong bisimulation equality.

Furthermore, the most important aspects of the implementation have been highlighted. For the implementation, numerous tools have been applied, e.g. for model-driven development (such as Eclipse Modelling Framework and Eclipse Xtext), or the Scala programming language (for building the LARES-core libraries) or external tools such as CASPA for performing the analysis. As a result, an Eclipse View-plugin emerged which allows managing experiments, fully automatic drawings of 2D plots for obtained measures or depicting the model-structure or state space.

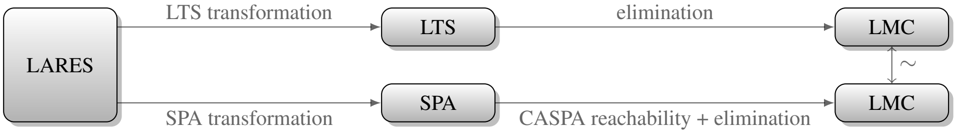


Figure 7.1: By definition, the LARES LTS and SPA transformation semantics results in two bisimulation equivalent labelled Markov chains (LMC)

Case-studies have also been presented to show the applicability of LARES. As part of that, scalability considerations have been made to clarify how well the analysis performs while increasing complexity within some aspects of the model, in particular regarding the number of product terms arising from guards, forward and Condition statements. This led to the ideas for future research questions on how to perform high-level abstractions/reductions on a LARES model (by using upper/lower bound techniques to achieve pessimistic/optimistic estimations) and how to exploit model structure and regularities. Another challenging future task is to find smart ways for distributing the analysis process of decomposed independent submodels. Furthermore, additional analysis methods may improve model construction or system design (such as sensitivity analysis, design space analysis or model checking).

In comparison to related approaches, LARES has unique characteristics, but still there is plenty of room for improvements. Various aspects could help improve the LARES language or toolset in order to become more expressive and feature-rich. The upcoming *reward extension* [70] and *decision extension* [69] have partially been realised, thereby providing an MDP semantics for LARES. The following subsections sketch some further directions of future research and development.

7.1 Language Extensions and Expressiveness

In addition to the LTS semantics defined for standard LARES or the MDP semantics that has been introduced by the *decision extension*, many other features can enrich the LARES language expressiveness, e.g. integrating arbitrary distributions, non-Boolean variables (to capture spacial, time and continuous aspects) in combination with giving users the opportunity to define more sophisticated expressions that go beyond standard Conditions which only depend on (Boolean) state variables, i.e. by allowing general first-order logic expressions that may also depend on non-Boolean variables.

Arbitrary Distributions

General probability distributions can be added to LARES without invalidating the current Markovian-based semantics by phase-type distributions. There are many algorithms which allow *fitting* such distributions by Markov processes comprising a minimal amount of states (e.g. as described in [116]). In order not to run into state space explosion by counting the performed phases, it might be better to categorically ensure by the given semantics that no interaction takes place until a phase-type distribution is absorbed (such that a behaviour instance is situated within an *atomic* sequence between the starting and the absorbing state of a phase-type distribution). Otherwise, this would result in unpredictable behaviour. Assume a phase-type distribution starting in (a_1, b) and absorbing in (a_2, b) such that $(a_1, b) \rightarrow (\_, b) \rightarrow (a_2, b)$ represents the Markov process until absorption. A guards statement such as $\neg\mathtt{a}_1$ `guards` $\langle ... \rangle$ could imply an invalid interruption of the phase-type distribution within intermediate phase $(\_, b)$. This could be avoided by introducing modifiers to hide state - or guard labels from a Behavior's environment.

Non-Boolean Variables

CASPA allows defining predicates on process parameters within guards as some sort of propositional logic that only allows integer arithmetic on the given parameters including relational operators to map into the Boolean domain, e.g. $[i > 5]$ with parameter i, to specify whether a certain transition can take place at a specific state defined by i. This feature is not yet employed by LARES. The specification of Condition statements on states is too cumbersome to mimic these predicates. Therefore, local non-Boolean (integer) variable statements which declare a name and some type or domain, e.g. $var\ x\ : [1..10]$ as well as an extended variant of Condition statements that allow reasoning about these variables in a kind of first-order logic notion could be introduced to the LARES language. As long as only integer variables are considered, these expressions can easily be captured by the CASPA process algebra via named processes or process parameters in cooperation with guarded processes.

- **Counting Variables for Deterministic Time.** Real-time aspects are also required by many modellers. Following the semantics as given for timed automata [13], a clock variable is considered a real typed variable and can be addressed by other primitives such as clock-invariants or guards. In order to comply with the above requirement, the LARES language has to be extended by primitives dealing with deterministic time, similarly to the MoDeST approach which relies on probabilistic timed automata [82].
- **Spatial Variables.** The introduction of real typed variables could be used to describe spatial information, for example how Module instances are distributed on a grid or a continuous area and, by doing so, to define how spatial information effects the subsequent behaviour. Let the spatial variables x, y for instance i_1 and i_2 be given, then a constraint such as $\sqrt{\Delta(i_1.x, i_2.x)^2 + \Delta(i_1.y, i_2.y)^2} \leq 3$ could be defined to check whether the Euclidean distance Δ of two instances is smaller or equal to 3. To illustrate, new notations of LARES statements are required to be introduced, e.g. to describe a rule for increasing some transmission power when two mobile ad-hoc nodes reach a specific distance:

$$(\sqrt{\Delta(\mathtt{i}_1.\mathtt{x}, \mathtt{i}_2.\mathtt{x})^2 + \Delta(\mathtt{i}_1.\mathtt{y}, \mathtt{i}_2.\mathtt{y})^2} \geq 3) \wedge \mathtt{i}_2.\mathtt{lowPower}\ \mathtt{guards}\ \mathtt{i}_2.\langle\mathtt{highPower}\rangle$$

- **Hybrid Systems.** Real-typed variables allow a wider range of physical systems to be described, wherein a state as well as its evolution can be continuous. For LARES, the extension of Condition statements on continuous domains (or even more general, the gradual change of a continuous state that affects the system's continuous or discrete behaviour) could be realised. Even now, guards statements define dependencies between states and subsequent behaviour. At present, guards statements are still of discrete manner, but they may also be extended to the continuous domain as stated previously. As such LARES would be enabled to be used as a language to model hybrid systems as well (e.g. following the concepts of the compositional Hybrid Input/Output Automaton approach, cf. [96]). Its evaluation remains difficult: Using a state-based representation for the analysis requires again a discretisation, but the resulting state space is usually intractable. As a consequence those kinds of models have to be handled differently, e.g. performing reachability predicate abstraction [6] or even incorporating SMT approaches [44].

7.2 Scalability Considerations

While language expressiveness only considers the ability to capture specific aspects of a model by language constructs, an issue arises from the formal semantics and the ability for its analysis. The semantics is based on a formal target language to which a model is transformed. It depends on the destination language whether the expressiveness of the source language may be captured. Otherwise, simulation techniques have to be applied, or subsets have to be addressed, or oversimplification is required, e.g. discretising a continuous variable, approximating immediate events by exponential distributions with high rates (which can lead to numerical instability), or approximating general stochastic distributions by phase-type distributions. As a result, the models may have an increased state space or it may be difficult to reveal how oversimplification impacts the analysis results. Therefore, it is of importance to consider the scalability of an approach regarding the ability to analyse an output model.

Reduced Synchronisation Combinatorics

Let a Module instance be composed of the subinstances A and B, a repair Behavior R and a guards statement inferring whether at least a single subinstance has failed (thereby it is irrelevant whether this occurred due to A or due to B) in order to invoke the repair process:

```
1 oo {A.⟨failed⟩, B.⟨failed⟩} guards R.⟨repair⟩
```

As long as R cannot perform the repair process, A or B can fail. If R reaches the state which can invoke the repair process, the repair will immediately take place whilst acting synchronously with actions of A or B which serve as triggers.

When abstracting from the sequence of the addressed variables, fewer action labels will be needed in order to achieve the required semantics of the guards statement by means of generated SPA processes. At present, different cases are captured by the SPA transformation semantics. Therefore, composed actions are built for the first case when only A has failed (indicated via parameter f) while B has not (indicated via parameter g) (such that the action oA can be performed), for the opposite case (when the action oB can be performed) and for the third case when both subinstances have failed (by the action b). These actions are used to trigger the repair process R to reach its repair state (indicated via parameter r):

```
(... + oA;A(f) + b;A(f)) |[oA, oB, b]| (... + oA;B(g))
  |[oA, oB, b]| (oA;R(r) + oB;R(r) + b;R(r))              // A failed
(... + oB;A(g)) |[b; oB]| (... + oB;B(f) + b;B(f))
  |[oA, oB, b]| (oA;R(r) + oB;R(r) + b;R(r))              // B failed
(... + oA;A(f) + b;A(f)) |[oA, oB, b]| (... + oB;B(f) + b;B(f))
  |[oA, oB, b]| (oA;R(r) + oB;R(r) + b;R(r))              // A & B failed
```

As stated above, it is irrelevant which combination (each encoded by an individual action) of subinstance failures triggers the event. The triggering of the repair process could instead be carried out by a single action enc for all cases:

```
((... + enc;A(f)) |[]| B(g)) |[enc]| enc;R(r)             // A failed
(A(g) |[]| (... + enc;B(f))) |[enc]| enc;R(r)             // B failed
((... + enc;A(f)) |[]| (... + enc;B(f))) |[enc]| enc;R(r) // A & B failed
```

It is desirable to generalise this for arbitrary guards statements in order to enable a further reduction of the number of encodings for a certain class of guards statements. However, the amount of how many encodings have to be stored heavily depends on the composition structure of the generated SPA model which currently strictly follows the hierarchy of the LARES model. The caveat is that for example two guards statements could require different composition structures in order to enable a minimal number of encodings for each of them, but which itself is impossible to realise in general. Heuristics might help find a good compromise. The problem is related to finding an adequate variable ordering in a BDD and requires additional computations to determine the composition structure of the arising processes. As a consequence, the system structure (as defined by the LARES model) will no longer be reflected by the composition structure of the generated SPA specification.

Other Reduction and Abstraction Techniques

As above, the use of non-Boolean variables or phase-type distributions may lead to an enormous growth of the number of states. To overcome this obstacle, several techniques were proposed and developed in the last decades which attract interest in future LARES developments. These approaches operate on different levels. They either work on language or the transformation level or they involve special low-level representations. CASPA, for example, symbolically encodes its transitions using MTBDD techniques on a very low-level. The composition operator is implemented such that the increase of size of the MTBDD is linear to the number composed processes (cf. Theorem 5.1.2 in [126]). Partial-Order-Reduction [65] and Confluence Reduction [80] represent two other low-level reduction methods. The first is state-based and the latter is transition-based. Other interesting approaches are described by so-called Magnifying Lens abstraction as described in [5, 122] which operate on a partition of the state space wherein regions are adaptively refined until an upper and a lower bound of a property reaches a predefined accuracy. These approaches are rather limited in their ability to reduce the state space since they operate on a state-based level. In contrast to these rather low-level state space reduction techniques, the structure and keywords offered by higher-level languages may directly be used to infer symmetric replications which allow producing a symmetry-reduced quotient model in order to keep track of "[...] how many processes reside in each local state [...]" [53]. A similar approach is the application of the *Mean Field Theory* to probabilistic systems by which many equivalent subcomponents can be approximated under the assumption that an infinite number of components provides a deterministic behaviour [28]. It only regards fractions of components being in a certain state. These fractions evolve in the course of time depending on the interactions among these fractions.

LTS Reachability

The reachability algorithm that constructs a LTS from a LARES$_{\text{FLAT}}$ model currently directly operates on the *case classes* representing the abstract language definition. For the purpose of checking bisimulation equivalence this approach is sufficient. In order to construct the state space of larger LARES models for the use of external Markov Chain solvers, the LARES$_{\text{FLAT}}$ model has to be encoded to exploit a better performance. Accordingly the TRA formalism as implemented in this work can not be used as a temporary structure to capture large state spaces. Instead, a compacted representation needs to be chosen in order to minimise the effort for the model exchange with the solver or to serve as a data structure on which performant algorithms can be defined for the analysis.

Statistical Analysis

A stochastic simulation engine for LARES could easily be derived from the reachability algorithm that works on the level of LARES$_{\text{FLAT}}$ to alleviate the state space explosion. For this purpose, the implemented reachability algorithm's state transition functionality can be reused. While the reachability algorithm explores each possible transition, the probability distribution of all possible transitions would be taken into account to determine the sojourn time of the current state. The transition for which the smallest value for the associated random variable is generated will be used to proceed to the next state. In order to analyse the measures of interest, statistics have to be collected. For this purpose, a clock variable is required and it needs to be updated within each step. Furthermore, a trace as a sequence of states has to be stored continuously which includes information taken from the clock variable and the transition taken. Nevertheless, especially dependable systems inherently deal with rare events. To efficiently capture these events, techniques of *rare event simulation* need to be incorporated (e.g. as done in [118]) to prevent excessive numbers of simulation runs. These techniques are commonly based on *variance reduction* such as *importance sampling* [38] or *importance splitting* (e.g. the restart method [136]). For future LARES extensions to capture hybrid systems, the level of granularity of the time intervals of discrete-event simulation may remain an issue regarding the evolution of continuous variables over time. Another option is to apply techniques and tools which have been developed to deal with hybrid system simulation. In order to give an example, a transformation into the hybrid χ-language (a process algebra for hybrid systems which binds tools for simulation as well as for numerical analysis [100]) could be defined. On the other hand, non-determinism is differently resolved depending on the applied simulator that can currently be addressed by the χ language, i.e. by priority, probabilistically, manually or even disallowed.

7.3 Modelling and Toolset

This section discusses improvements which aim at supporting a modeller in the process of defining, validating or analysing a model. Therefore, user feedback on system properties, experiment management or hybrid editing capabilities could be integrated into the LARES IDE.

User Feedback by proving System Properties

Overlapping conditions can be specified by denoting a number of guards statements so that several conditions may simultaneously be satisfied by a composed state. According to that, several events could occur in form of a choice. A modeller's task is to carefully partition the conditions in order to avoid unwanted choices among competing transitions. Currently, there is no support to warn modellers whether such a competing situation is introduced to the model. To help modellers with detecting such modelling flaws, a kind of debugging tool could be integrated or developed in the future. This can be carried out by either applying tools for model checking or by implementing a simple structural analysis within CASPA in order to search for multiple outgoing competing transitions for each state after performing a composition. A feedback mechanism has to be implemented in the LARES IDE to warn users when they inadvertently introduce choices.

The process of modelling this kind of feedback cannot be performed on-the-fly for larger submodels. The reason is that the composition time will be an issue regarding the responsiveness of the modelling environment. For larger models this kind of validation has to be performed off-the-fly, but it should be performed on a regular basis (i.e. whenever new guards statements are introduced).

Experiments Management Capabilities

The View-Plugin developed as part of the LARES Eclipse environment provides the capabilities to carry out the analysis process and to visualise the model in different views or the analysis results. However, the feature to manage experiments has not yet been implemented. For that purpose, a data-base binding would be useful to capture the different model versions and measures calculated thereof.

Hybrid Graphical/Textual Editor

The editor has improved a lot since its first version. Standard Xtext features such as *code completion* and *syntax highlighting* have been refined. Furthermore, sophisticated on-the-fly and standard validation capabilities have been integrated. Different LARES meta-models have hence been defined and applied to perform the required validation calculations implemented by MDD model transformation techniques using Eclipse Epsilon [55]. Another matter is the development of a graphical editor applying Graphiti [56], an EMF based graphical tooling infrastructure. Noticeable progress has been made to represent most of the LARES features by two diagram types defined for Behavior and Module definitions. The long-term goal is to offer a coherent hybrid LARES Editor Plugin which allows switching between textual and graphical representations in order to give modellers the opportunity to choose for the most suitable representation from both worlds.

7.4 Closing Comment

Even though the LARES framework is an outcome of an academic research project, it could serve as a basis for further developments which contribute to a productive environment for modelling dependable, reconfigurable systems. The transformation process is already very mature and can be adapted to future language extensions without too much effort. At present time two extensions have almost completely been implemented allowing the definition of rewards and non-deterministic decisions. The extensions of the transformations went smoothly and proved the flexibility of the selected implementation approach. A website [50] has been set up to promote the LARES environment, to provide publications related to LARES, to offer tutorials, to invite others to join the development by providing them with the source code or allow them to apply the LARES toolset by providing platform specific binaries. It now depends on future development efforts to keep LARES a vividly emerging dependability modelling language.

Appendix A

Frequently Used Attributed Tuples

Identifier	$(l, I) \in \mathfrak{ID}$
Reference	$(i, l) \in \mathfrak{Ref}$
Parameter expression	$\{(l_1, e_1), \ldots\} \in \mathfrak{PE}$
Definition reference	$(r, p) \in \mathfrak{Ref}_{\mathcal{D}}$
Behavior definition	$(l, p, b, ic) \in \mathcal{B}$
Behavior body	$(S, T, E, ic) \in \mathfrak{B}_{body}$
Transitions	$(s, g, d, t) \in \mathcal{T}$
Distribution (Immediate/Markovian)	$(type, value) \in \mathcal{D}$
Behavior expand statement	$(ie, b) \in \mathfrak{B}_{expand}$
Module definition	$(l, p, d, b) \in \mathcal{M}$
Module body	$(B, M, I, C, G, F, IC, E, P) \in \mathfrak{M}_{body}$
Instance statement	$(l, t, ic) \in \mathcal{I}$
Condition statement	$(l, c) \in \mathcal{C}$
guards statement	$(g, CR, ns) \in \mathcal{G}$
forward statement	$(c, l, CR) \in \mathcal{F}$
Conditional reactive	$(c, r) \in^k \mathfrak{CR}$
Initial statement	$(l, IC) \in \mathfrak{IC}$
Probability statement (steady-state)	$(l, c) \in \mathfrak{Prob}_S$
Probability statement (transient-state)	$(l, c, t) \in \mathfrak{Prob}_T$
Module expand statement	$(ie, b) \in \mathfrak{M}_{expand}$
Guard label reference	$(i, l, dt) \in \mathfrak{Ref}_R$
Iterator	$(l, s) \in \mathfrak{I}$

A FREQUENTLY USED ATTRIBUTED TUPLES

Appendix B

Common Substitution and Evaluation Functions

The following function declaration were used in Section 3.3.1 without any given definition:

- Set Expression Resolution:
$^{\mathfrak{h}}\!/\!\phi^{\mathfrak{SE}} : \mathfrak{SE} \times \mathfrak{PE} \rightarrow \mathfrak{SE}$
- Condition Expression Resolution:
$^{\mathfrak{h}}\!/\!\phi^{\mathfrak{CE}} : \mathfrak{CE} \times \mathfrak{PE} \rightarrow \mathfrak{CE}$
- Reactive Expression Resolution:
$^{\mathfrak{h}}\!/\!\phi^{\mathfrak{RE}} : \mathfrak{RE} \times \mathfrak{PE} \rightarrow \mathfrak{RE}$
- Identifier Resolution:
$^{\mathfrak{h}}\!/\!\phi^{\mathfrak{ID}} : \mathfrak{ID} \times \mathfrak{PE} \rightarrow \mathfrak{ID}$
- Reference Resolution:
$^{\mathfrak{h}}\!/\!\phi^{\mathfrak{Ref}} : \mathfrak{Ref} \times \mathfrak{PE} \rightarrow \mathfrak{Ref}$
- Arithmetic Expression Resolution:
$^{\mathfrak{h}}\!/\!\phi^{\mathfrak{AE}} : \mathfrak{AE} \times \mathfrak{PE} \rightarrow \mathfrak{AE}$
- Set Expression Evaluation:
$\phi^{\mathfrak{SE}} : \mathfrak{SE} \rightarrow \mathcal{P}(\mathbb{R})$
- Parameter Expression Evaluation:
$\phi^{\mathfrak{PE}} : \mathfrak{PE} \rightarrow \mathfrak{PE}$

This is caught up with the following definitions. The parameter resolution function $^{\mathfrak{h}}\!/\!\phi^{\mathfrak{SE}}$ for a set expression is therefore defined as follows (op hereby denotes an operator of a set expression and ae_1 and ae_2 are arithmetic expressions):

$$^{\mathfrak{h}}\!/\!\phi^{\mathfrak{SE}} : se, pe \mapsto \begin{cases} (^{\mathfrak{h}}\!/\!\phi^{\mathfrak{SE}}(se_1, pe), op, {}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{SE}}(se_2, pe)) & \text{if } se = (se_1, op, se_2) \\ (^{\mathfrak{h}}\!/\!\phi^{\mathfrak{AE}}(ae_1, pe), {}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{AE}}(ae_2, pe)) & \text{if } se = (ae_1, ae_2) \\ \left\{ {}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{AE}}(ae, pe) \mid ae \in se \right\} & \text{if } se \in \mathcal{P}(\mathfrak{AE}) \end{cases}$$

The parameter resolution function ${}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{AE}}$ for an arithmetic expression is defined as follows (op hereby denotes an operator of an arithmetic expression and ae_1 and ae_2 are arithmetic expressions):

$${}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{AE}} : ae, pe \mapsto \begin{cases} ({}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{AE}}(ae_1, pe), op, {}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{AE}}(ae_2, pe)) & \text{if } ae = (ae_1, op, ae_2) \\ ae & \text{if } ae \in \mathbb{R} \\ e & \text{if } ae \in \Sigma^* \wedge \exists (l, e) \in pe : ae = l \end{cases}$$

The parameter resolution function ${}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{ID}}$ for an identifier is defined as follows:

$${}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{ID}} : id, pe \mapsto \begin{cases} (l, ({}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{AE}}(ae_1, pe), \ldots)) & \text{if } id = (l, (ae_1, \ldots)) \\ (l, ()) & \text{else if } id = (l, ()) \end{cases}$$

The parameter resolution function ${}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{Ref}}$ for a reference is defined as follows:

$${}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{Ref}} : ref, pe \mapsto \begin{cases} (({}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{ID}}(inst, pe)), {}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{ID}}(l, pe)) & \text{if } ref = ((inst), l) \\ ((), {}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{ID}}(l, pe)) & \text{else if } id = ((), l) \end{cases}$$

The parameter resolution function ${}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{CE}}$ for a condition expression is defined as follows:

$${}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{CE}} : ce, pe \mapsto \begin{cases} ({}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{CE}}(ce_1, pe), op, {}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{CE}}(ce_2, pe)) & \text{if } ce = (ce_1, op, ce_2) \end{cases}$$

The parameter resolution function ${}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{RE}}$ for a reactive expression is defined as follows:

$${}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{RE}} : re, pe \mapsto \begin{cases} sync({}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{RE}}(re_1, pe), \ldots) & \text{if } re = sync(re_1, \ldots) \\ maxsync({}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{RE}}(re_1, pe), \ldots) & \text{if } re = maxsync(re_1, \ldots) \\ choose({}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{RE}}(re_1, pe), \ldots) & \text{if } re = choose(re_1, \ldots) \\ {}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{Ref}}(re, pe) & \text{if } re \in \mathfrak{Ref} \end{cases}$$

The evaluation function $\phi^{\mathfrak{AE}}$ for an arithmetic expression is defined so that it can be used by the subsequent definitions (op hereby denotes an operator of an arithmetic expression and ae_1 and ae_2 are arithmetic expressions):

$$\phi^{\mathfrak{AE}} : ae \mapsto \begin{cases} [\![{}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{AE}}(ae_1, pe) \; op \; {}^{\mathfrak{h}}\!/\!\phi^{\mathfrak{AE}}(ae_2, pe)]\!] & \text{if } ae = (ae_1, op, ae_2) \\ [\![ae]\!] & \text{if } ae \in \mathbb{R} \end{cases}$$

The evaluation function $\phi^{\mathfrak{SE}}$ of a set expression (*op* hereby denotes an operator of a set expression) is defined as follows:

$$\phi^{\mathfrak{SE}} : se \mapsto \begin{cases} [\![\phi^{\mathfrak{AE}}(se_1)\ op\ \phi^{\mathfrak{AE}}(se_2)]\!] & \text{if } se = (se_1, op, se_2) \\ \bigcup_{i=\phi^{\mathfrak{AE}}(ae_1)}^{\phi^{\mathfrak{AE}}(ae_2)} \{i\} & \text{if } se = (ae_1, ae_2) \\ \left\{\phi^{\mathfrak{AE}}(ae) \mid ae \in se\right\} & \text{if } se \in \mathcal{P}(\mathfrak{AE}) \end{cases}$$

Finally, the evaluation function $\phi^{\mathfrak{PE}}$ of a parameter expression is defined:

$$\phi^{\mathfrak{PE}} : pe \mapsto \{\,(p.l, \phi^{\mathfrak{AE}}(p.e)) \mid p \in pe\}$$

Appendix C

Refined Synchronisation Semantics of PACT

In Section 3.5.3 solely the essentials of the PACT synchronisation which are needed for the transformation from LARES$_{\text{BASE}}$ into SPA are described. A more detailed explanation is given hereinafter.

The information $S \in \mathcal{S}$ on how a process interacts with its environment is defined as a triple of sets of tuples (where each tuple comprises an action label and an associated distribution type):

$$\mathcal{S} = \mathcal{P}(Act \times D_{types})^3 \text{ , where } (A_c, A_s, A_n) \in \mathcal{S}.$$

The attribute A_c denotes the set of actively consuming actions (i.e. actions which, if synchronised with another process, will not be needed for further synchronisation of the resulting composed process), A_s denotes the set of passively synchronising actions (i.e. actions which allow being further synchronised) and A_n represents the set of actions that are required to stay unsynchronised (and may be used to detect whether an instance specifies disallowed synchronisations).

A parallel composition operation $||: \mathcal{PAC} \times \mathcal{PAC} \rightarrow \mathcal{PAC}$ will be defined to construct valid SPA composition terms (which may also make use of the hide operator), to determine the corresponding synchronisation information and to return the related process definitions.

When a composition of two PAC structures $l, r \in \mathcal{PAC}$ takes place, the set of consumed actions (i.e. actions for which a consuming partner can be found) is given by

$$A_{consumed} = (l.S.A_c \cap r.S.A_s) \cup (r.S.A_c \cap l.S.A_s) \cup (r.S.A_c \cap l.S.A_c)$$

In addition, the set of synchronising action labels is composed of the set of consumed action labels and by those labels of passively synchronising actions for which a non-consuming partner can be found:

$$A_{synched} = \{\pi_2(act) \mid act \in ((l.S.A_s \cap r.S.A_s) \cup A_{consumed})\}$$

By the fact that solely immediate-typed actions can be hidden (with regard to the CASPA semantics), action labels (which have an immediate-typed distribution and satisfy an implementation specific control function ϕ) are taken from the set of *hidable actions* which consists of consumed actions and actions that are required to stay unsynchronised:

$$A_{hide} = \{\, a \mid \overset{a}{\dashrightarrow} \in (\underbrace{A_{consumed} \cup l.S.A_N \cup r.S.A_N}_{\text{hidable actions}}) : \phi(\ldots)\}$$

The set of consuming action labels A_c, the set of synchronising action labels A_s and the set of action labels remaining unsynchronised A_n are determined as follows:

$$\begin{aligned} A_c &= (l.S.A_c \cup r.S.A_c) \setminus (A_{consumed} \cup A_{hide}) \\ A_s &= (l.S.A_s \cup r.S.A_s) \setminus (A_{consumed} \cup A_{hide}) \\ A_n &= (l.S.A_n \cup r.S.A_n \cup A_{consumed}) \setminus A_{hide} \end{aligned}$$

The definition of the parallel composition function $||$ is therefore given as follows:

$$||: (l, r) \mapsto \big(\underbrace{\text{ hide } A_{hide} \text{ in } (l.t|[A_{synched}]|r.t)}_{\text{process algebra term}}, \underbrace{(A_c, A_s, A_n)}_{\text{sync. info.}}, \underbrace{(l.D \cup r.D)}_{\text{proc. definitions}}\big)$$

Let an n-ary parallel composition function be given by $||^n : \mathcal{P}(\mathcal{PAC}) \to \mathcal{PAC}$ which makes use of the binary composition to create a binary composition tree of processes. The final specification is constructed by the function $pact2spa : PACTN \to PAT$.

It recursively processes the PACT structure such that t finally represents the initial process name and D consists of all process definitions which encompass the full SPA specification of the model:

$$\begin{aligned} &pact2spa : e \mapsto (e.x, S, D \cup \{e.x := t\}), \text{ where} \\ &\quad (t, S, D) = ||^n(\underbrace{\{pact2spa(c) \mid c \in e.C \cap \mathcal{PACN}\}}_{\text{substructures}} \cup \underbrace{(e.C \cap \mathcal{PAT})}_{\text{leaves}}) \end{aligned}$$

Appendix D

On Combining Generative/Reactive Expressions

This section describes how generative and reactive expressions are combined by transforming them into their normal forms, so that a distinct behaviour is obtained regarding the composed state space.

Literals, DNF, Canonical DNF

Definition 1 (Literal)**.** *A literal is an expression of the form x or $\neg x$, where x is a Boolean variable either representing a proposition on a state or on a guarded transition.*

Definition 2 (DNF)**.** *A disjunctive normal form is defined as an expression of the form*

$$\bigvee_{k=1}^{m} C_k = \bigvee_{k=1}^{m} \left(\bigwedge_{i \in A_k} x_i \wedge \bigwedge_{j \in B_k} \neg x_j \right) \tag{D.1}$$

where each $C_k (k = 1, 2, \ldots, m)$ is an elementary conjunction, i.e. a product term of the DNF.

Definition 3 (Canonical DNF)**.** *Let f be a Boolean function on B^n, let $T(f)$ be the set of true evaluations of f. A minterm of f is an elementary conjunction of the form $\left(\bigwedge_{i|y_i=1} x_i \wedge \bigwedge_{j|y_j=0} \neg x_j\right)$, where $Y = (y_1, \ldots, y_n) \in T(f)$. The DNF*

$$\phi_f(x_1, \ldots, x_n) = \bigvee_{Y \in T(f)}^{m} \left(\bigwedge_{i|y_i=1} x_i \wedge \bigwedge_{j|y_j=0} \neg x_j \right) \tag{D.2}$$

is the canonical DNF of f.

Minterms can be uniquely numbered by using a binary encoding for a given variable order (e.g. an alphabetical order). Let an unnegated literal be assigned with value 1 and a negated literal with value 0. For example, $a\bar{b}c$ is encoded by the value 101_2 in binary system or 5_{10} in decimal system. Standard product terms can also be numbered as above. Therefore, a fixed variable order is needed and *don't care* variables are valued by e.g. 0.

The Exhaustive Approach

Let the instance tree depicted in Figure D.1 be given. It has the following guards statement inside the black-filled node:

```
L.L.B.a | L.B.b guards maxsync {L.B.⟨c⟩, R.R.B.⟨d⟩}
```

Let ns be the current namespace and let the conditional reactive arising from the above guards statement be enumerated by 0. Both the generative and the reactive part are Boolean expressions which can be represented by their *canonical DNF* (consisting of minterms which represent satisfiable paths). A function $pt_{minterm}$ is declared which determines and enumerates all (distinct) canonical minterms for a given Boolean expression:

$$pt_{minterm} : \mathfrak{BE} \rightarrow \mathcal{P}(\mathfrak{BE} \times \mathbb{N})$$

There may be different definitions implemented which are not further detailed.

A Boolean expression will be fulfilled if one of the arising product terms is fulfilled (see Eq. (D.1)). In LARES, a product term combination where both product terms are fulfilled will enable a transition in the composed system. As illustrated in Table D.1, a composed enumeration is built which includes ns and 0 in order to derive a unique encoding for each minterm combination by building the cross-product of both sets of enumerated minterms.

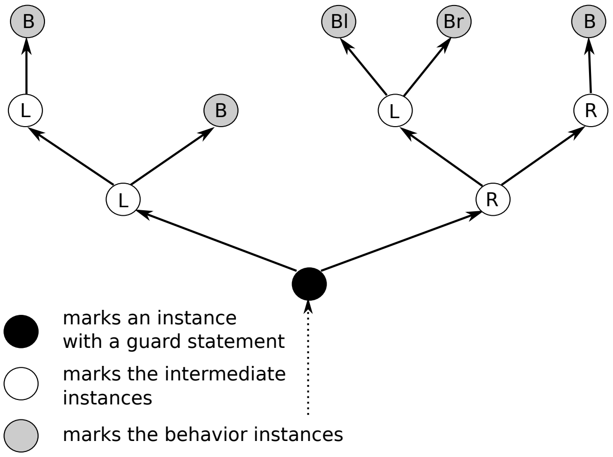

B
Bl
Br
B
L
B
L
R
L
R
marks an instance with a guard statement
marks the intermediate instances
marks the behavior instances

Figure D.1: Instance subtree from the perspective of a single guards statement

generative part	reactive part	encoding
$L.L.B.a \vee L.B.b$	$L.B.\langle c\rangle \vee R.R.B.\langle d\rangle$	ns,0,_,_
$\Downarrow pt_{minterm}(L.L.B.a \vee L.B.b)$		
$L.L.B.a \wedge L.B.b$	$L.B.\langle c\rangle \vee R.R.B.\langle d\rangle$	ns,0,0,_
$L.L.B.a \wedge \neg L.B.b$	$L.B.\langle c\rangle \vee R.R.B.\langle d\rangle$	ns,0,1,_
$\neg L.L.B.a \wedge L.B.b$	$L.B.\langle c\rangle \vee R.R.B.\langle d\rangle$	ns,0,2,_
	$\Downarrow pt_{minterm}(L.B.\langle c\rangle \vee R.R.B.\langle d\rangle)$	
$L.L.B.a \wedge L.B.b$	$L.B.\langle c\rangle \wedge R.R.B.\langle d\rangle$	ns,0,0,0
$L.L.B.a \wedge L.B.b$	$L.B.\langle c\rangle \wedge \neg R.R.B.\langle d\rangle$	ns,0,0,1
$L.L.B.a \wedge L.B.b$	$\neg L.B.\langle c\rangle \wedge R.R.B.\langle d\rangle$	ns,0,0,2
$L.L.B.a \wedge \neg L.B.b$	$L.B.\langle c\rangle \wedge R.R.B.\langle d\rangle$	ns,0,1,0
$L.L.B.a \wedge \neg L.B.b$	$L.B.\langle c\rangle \wedge \neg R.R.B.\langle d\rangle$	ns,0,1,1
$L.L.B.a \wedge \neg L.B.b$	$\neg L.B.\langle c\rangle \wedge R.R.B.\langle d\rangle$	ns,0,1,2
$\neg L.L.B.a \wedge L.B.b$	$L.B.\langle c\rangle \wedge R.R.B.\langle d\rangle$	ns,0,2,0
$\neg L.L.B.a \wedge L.B.b$	$L.B.\langle c\rangle \wedge \neg R.R.B.\langle d\rangle$	ns,0,2,1
$\neg L.L.B.a \wedge L.B.b$	$\neg L.B.\langle c\rangle \wedge R.R.B.\langle d\rangle$	ns,0,2,2

Table D.1: Illustration of the encoding of all tuple combinations

All combinations from the given guards statement can thus be encoded using $pt_{minterm}$. The first column of the figure considers the generative part of the guards statement, whereas the second column considers the reactive part. The third column contains the derived encoding. Hereby, the application of $pt_{minterm}$ on the generative part yields a block of rows representing the product due to the minterms and the additional encoding information. The application of $pt_{minterm}$ on the reactive part finally completes all minterm combinations and their encoding.

Towards A More Efficient Encoding

It turns out that the number of encodings can safely be reduced by not explicitly generating a canonical DNF representation for the generative part. Instead, addressed state variables of Behavior instances may be omitted by merging minterms. By doing so, a sum of product terms arises with fewer product terms and fewer variables within the merged terms. A DNF is suitable in case it has a minimal number of product terms (and a minimal number of literals). BDDs are used in order to find a compactified representation. In [21] it is shown that finding an optimal solution is an NP-complete problem. Depending on the variable ordering a BDD can vary between being exponential or polynomial in size of the number of variables. In [78] a number of algorithms finding optimal variable orderings is given. However, it is far beyond the scope of this work to go into details of these algorithms.

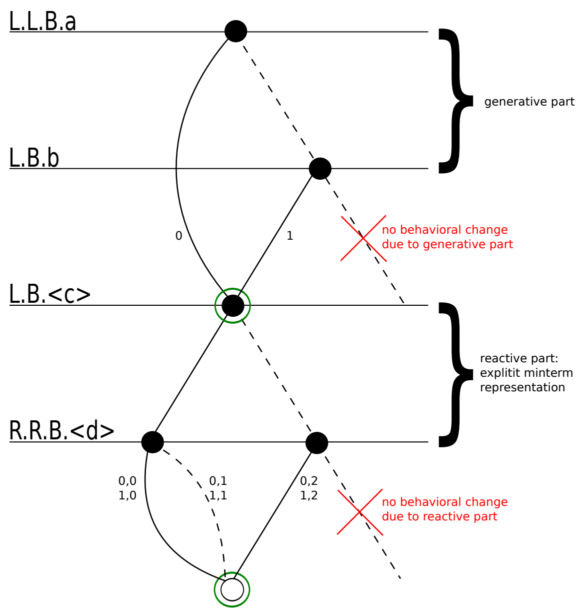


Figure D.2: Using BDDs to obtain simplified DNFs

generative part	reactive part	encoding
$L.L.B.a \vee L.B.b$	$L.B.\langle c\rangle \vee R.R.B.\langle d\rangle$	ns,0,-,-
$\Downarrow pt(L.L.B.a \vee L.B.b)$		
$L.L.B.a$	$L.B.\langle c\rangle \vee R.R.B.\langle d\rangle$	ns,0,0,-
$\neg L.L.B.a \wedge L.B.b$	$L.B.\langle c\rangle \vee R.R.B.\langle d\rangle$	ns,0,1,-
	$\Downarrow pt_{minterm}(L.B.\langle c\rangle \vee R.R.B.\langle d\rangle)$	
$L.L.B.a$	$L.B.\langle c\rangle \wedge R.R.B.\langle d\rangle$	ns,0,0,0
$L.L.B.a$	$L.B.\langle c\rangle \wedge \neg R.R.B.\langle d\rangle$	ns,0,0,1
$L.L.B.a$	$\neg L.B.\langle c\rangle \wedge R.R.B.\langle d\rangle$	ns,0,0,2
$\neg L.L.B.a \wedge L.B.b,$	$L.B.\langle c\rangle \wedge R.R.B.\langle d\rangle$	ns,0,1,0
$\neg L.L.B.a \wedge L.B.b,$	$L.B.\langle c\rangle \wedge \neg R.R.B.\langle d\rangle$	ns,0,1,1
$\neg L.L.B.a \wedge L.B.b,$	$\neg L.B.\langle c\rangle \wedge R.R.B.\langle d\rangle$	ns,0,1,2

Table D.2: Illustration of the encoding of all tuple combinations

The function $pt : \mathfrak{BE} \rightarrow \mathcal{P}(\mathfrak{BE} \times \mathbb{N})$ which uses BDDs in order to generate the product terms and enumerate them is accordingly declared. Note that the function $pt_{minterm}$ can also be implemented by using BDDs. This requires the function to explicitly consider *don't care* variables (by their true and false evaluation) in order to determine all minterms of the canonical DNF. A composed application of pt on the generative part and $pt_{minterm}$ on the reactive part is shown in Figure D.2 as a single BDD using a fixed variable ordering. Fewer satisfiable paths arise (as listed in Table D.2) by doing so than by using canonical DNFs (cf. Table D.1).

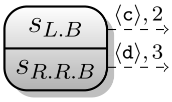
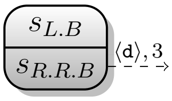
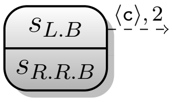
Revealed Behaviour	Canonical minterms by using $pt_{minterm}$: $\underbrace{\texttt{L.B.}\langle\texttt{c}\rangle\texttt{R.R.B.}\langle\texttt{d}\rangle}_{mt_{e1}} \vee \underbrace{\neg\texttt{L.B.}\langle\texttt{c}\rangle\texttt{R.R.B.}\langle\texttt{d}\rangle}_{mt_{e2}} \vee \underbrace{\texttt{L.B.}\langle\texttt{c}\rangle\neg\texttt{R.R.B.}\langle\texttt{d}\rangle}_{mt_{e3}}$	Non-canonical product terms by using pt: $\underbrace{\texttt{L.B.}\langle\texttt{c}\rangle\neg\texttt{R.R.B.}\langle\texttt{d}\rangle}_{pt_{e1}} \vee \underbrace{\texttt{R.R.B.}\langle\texttt{d}\rangle}_{pt_{e2}}$
	$\dfrac{s\models mt_{e1} \quad s\not\models mt_{e2} \quad s\not\models mt_{e3}}{(s_{L.B},s_{R.R.B})\overset{e1,6}{\dashrightarrow}(s'_{L.B},s'_{R.R.B})}$	$\dfrac{s\not\models pt_{e1} \quad s\models pt_{e2}}{(s_{L.B},s_{R.R.B})\overset{e2,3}{\dashrightarrow}(s_{L.B},s'_{R.R.B})}$
	$\dfrac{s\not\models mt_{e2} \quad s\models mt_{e2} \quad s\not\models mt_{e3}}{s\overset{e2,3}{\dashrightarrow}\ldots}$	$\dfrac{s\not\models pt_{e1} \quad s\models pt_{e2}}{s\overset{e2,3}{\dashrightarrow}\ldots}$
	$\dfrac{s\not\models mt_{e3} \quad s\not\models mt_{e2} \quad s\models mt_{e3}}{s\overset{e3,2}{\dashrightarrow}\ldots}$	$\dfrac{s\models pt_{e1} \quad s\not\models pt_{e2}}{s\overset{e1,2}{\dashrightarrow}\ldots}$

Table D.3: A comparison of the outcome when applying the SPA semantics for LARES by using product terms constructed via an BDD with variable order $R.R.B.\langle\texttt{d}\rangle < L.B.\langle\texttt{c}\rangle$ or by using canonical minterms instead (hereby, $s := (s_{L.B}, s_{R.R.B})$)

The reactive expression maxsync$\{$L.B.$\langle$c$\rangle$, R.R.B.$\langle$d$\rangle\}$ is used to illustrate why applying the function pt for transforming reactive expressions into non-canonical product terms does not match the intended semantics of LARES. In Table D.3 three cases are distinguished. Hereby, the first column illustrates the starting situation of each case with regard to the ability of the Behavior instances $L.B$ and $R.R.B$ to perform a guarded transition $\langle$c$\rangle$ and $\langle$d$\rangle$, respectively. The second column applies $pt_{minterm}$ in order to generate canonical minterms as implemented in the LARES framework. The third column shows which events would occur if non-canonical product terms were constructed using the function pt. Hereby, the first case leads to incorrect semantics as it differs in the obtained weight and the composed target state. Beyond that, the kind of arising weight and target states by using non-canonical product terms is ambiguous and heavily depends on the applied variable order. For these reasons, canonical minterms have to be used for the reactive part, where the application of $p_{minterm}$ is unambiguous and coincides with the intended LARES semantics.

Bibliography

[1] Aceto, L., Fokkink, W., Verhoef, C.: Structural Operational Semantics. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) Handbook of Process Algebra, pp. 197–292. Elsevier Science (2001), `http://dx.doi.org/10.1016/B978-044482830-9/50021-7` (Cited on page 94)

[2] Adeku, M.: A Study of Multiset Algebras: A Systematization of Fundamentals of Multiset Theory. Lambert Academic Publishing (2011) (Cited on page 27)

[3] Adeline, R., Cardoso, J., Darfeuil, P., Humbert, S., Seguin, C.: Toward a methodology for the AltaRica modelling of multi-physical systems. In: Ale, B., Papazoglou, I., Zio, E. (eds.) Proceedings of European Safety and Reliability Conference, ESREL 2010. Rhodes (Greece) (September 2010) (Cited on page 190)

[4] Aizpurua, J.I., Muxika, E.: Model-Based Design of Dependable Systems: Limitations and Evolution of Analysis and Verification Approaches. International Journal on Advances in Security (IARIA Conferences) 6(1&2), 12–31 (2013), `http://www.iariajournals.org/security/tocv6n12.html` (Cited on page 194)

[5] Alfaro, L., Roy, P.: Magnifying-Lens Abstraction for Markov Decision Processes. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 4590, pp. 325–338. Springer (2007), `http://dx.doi.org/10.1007/978-3-540-73368-3_38` (Cited on page 200)

[6] Alur, R., Dang, T., Ivančic, F.: Progress on Reachability Analysis of Hybrid Systems Using Predicate Abstraction. In: Maler, O., Pnueli, A. (eds.) Hybrid Systems: Computation and Control, Lecture Notes in Computer Science, vol. 2623, pp. 4–19. Springer Berlin Heidelberg (2003), `http://dx.doi.org/10.1007/3-540-36580-X_4` (Cited on page 198)

[7] Ammar, H., Huang, Y., Liu, R.: Hierarchical models for systems reliability, maintainability, and availability. Circuits and Systems, IEEE Transactions on 34(6), 629–638 (Jun 1987) (Cited on page 107)

[8] Arnold, A.: Mec: a system for constructing and analysing transition systems. In: Sifakis, J. (ed.) Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science, vol. 407, pp. 117–132. Springer (1990), http://dx.doi.org/10.1007/3-540-52148-8_11 (Cited on page 190)

[9] AT&T Research: Graph Visualization Software (2007), http://www.graphviz.org, [Online; accessed 3-September-2013] (Cited on page 138)

[10] Bachmann, J.: Entwurf und Implementierung eines graphischen Modelleditors und einer Benutzerschnittstelle für das Werkzeug CASPA. Master's thesis, Universität der Bundeswehr München, Dept. of Computer Science 4 (in German) (2007) (Cited on page 137)

[11] Bachmann, J., Riedl, M., Schuster, J., Siegle, M.: An Efficient Symbolic Elimination Algorithm for the Stochastic Process Algebra Tool CASPA. In: Nielsen, M., Kučera, A., Miltersen, P., Palamidessi, C., Tůma, P., Valencia, F. (eds.) SOFSEM 2009: Theory and Practice of Computer Science, Lecture Notes in Computer Science, vol. 5404, pp. 485–496. Springer (2009), http://dx.doi.org/10.1007/978-3-540-95891-8_44 (Cited on pages 58, 103, 104, 106, and 137)

[12] Behrmann, G., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Developing UPPAAL over 15 Years. Software: Practice and Experience 41(2), 133–142 (February 2011), http://dx.doi.org/10.1002/spe.1006 (Cited on page 187)

[13] Bengtsson, J., Yi, W.: Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets, Lecture Notes in Computer Science, vol. 3098, pp. 87–124. Springer (2004), http://dx.doi.org/10.1007/978-3-540-27755-2_3 (Cited on page 197)

[14] Berthomieu, B., Bodeveix, J.P., Farail, P., Filali, M., Garavel, H., Gaufillet, P., Lang, F., Vernadat, F.: Fiacre: An Intermediate Language for Model Verification in the Topcased Environment. In: ERTS 2008. Toulouse, France (2008), http://hal.inria.fr/inria-00262442 (Cited on page 194)

[15] Blum, M., Schiller, F.: Effiziente Sicherheitsmodellierung in der Automatisierungstechnik. In: Verl, A., Bender, K., Schumacher, W. (eds.) Tagungsband SPS/IPC/-DRIVES 2009. pp. 189–197. VDE Verlag GmbH, Berlin (2009) (Cited on page 191)

[16] Bogdoll, J., David, A., Hartmanns, A., Hermanns, A.: mctau: Bridging the Gap between Modest and UPPAAL. In: Donaldson, A., Parker, D. (eds.) Model Checking Software, Lecture Notes in Computer Science, vol. 7385, pp. 227–233. Springer (2012), http://dx.doi.org/10.1007/978-3-642-31759-0_16 (Cited on page 187)

[17] Bogdoll, J., Hartmanns, A., Hermanns, H.: Simulation and Statistical Model Checking for Modestly Nondeterministic Models. In: Schmitt, J.B. (ed.) Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance, Lecture Notes in Computer Science, vol. 7201, pp. 249–252. Springer (2012), http://dx.doi.org/10.1007/978-3-642-28540-0_20 (Cited on page 187)

[18] Bohnenkamp, H., D'Argenio, P., Hermanns, H., Katoen, J.: MODEST: A Compositional Modeling Formalism for Hard and Softly Timed Systems. IEEE Transactions on Software Engineering 32(10), 812–830 (2006), http://dx.doi.org/10.1109/TSE.2006.104 (Cited on page 187)

[19] Bohnenkamp, H., Hermanns, H., Katoen, J.P.: MOTOR: The MODEST Tool Environment. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 4424, pp. 500–504. Springer (2007), http://dx.doi.org/10.1007/978-3-540-71209-1_38 (Cited on page 187)

[20] Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S.: Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications. Wiley, 2nd edn. (2006) (Cited on page 1)

[21] Bollig, B., Wegener, I.: Improving the Variable Ordering of OBDDs is NP-Complete. IEEE Transactions on Computers 45, 993–1002 (September 1996), http://dx.doi.org/10.1109/12.537122 (Cited on page 215)

[22] Boudali, H., Crouzen, P., Haverkort, B.R., Kuntz, M., Stoelinga, M.: Arcade – A Formal, Extensible, Model-based Dependability Evaluation Framework. In: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2008. pp. 243–248. IEEE Computer Society Press, Los Alamitos (April 2008), http://doc.utwente.nl/64850/ (Cited on pages 181, 182, and 184)

[23] Boudali, H., Crouzen, P., Haverkort, B.R., Kuntz, M., Stoelinga, M.: Rich Interfaces for Dependability: Compositional Methods for Dynamic Fault Trees and Arcade models. In: Proceedings of the Second Workshop on Foundations of Interface Theories (FIT 2008). pp. 5–10. University of Aalborg, Aalborg, Denmark (April 2008), http://doc.utwente.nl/65424/ (Cited on page 182)

[24] Boudali, H., Crouzen, P., Haverkort, B., Kuntz, M., Stoelinga, M.: Architectural dependability evaluation with Arcade. In: Proceedings of the IEEE International Conference Dependable Systems and Networks With FTCS and DCC, DSN 2008.

pp. 512–521. IEEE Computer Society Press, Los Alamitos (2008), http://doc.utwente.nl/64537/ (Cited on pages 182 and 183)

[25] Boudali, H., Crouzen, P., Stoelinga, M.: A Compositional Semantics for Dynamic Fault Trees in Terms of Interactive Markov Chains. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) Automated Technology for Verification and Analysis, Lecture Notes in Computer Science, vol. 4762, pp. 441–456. Springer (2007), http://dx.doi.org/10.1007/978-3-540-75596-8_31 (Cited on page 182)

[26] Boudali, H., Crouzen, P., Stoelinga, M.: Dynamic Fault Tree Analysis Using Input/Output Interactive Markov Chains. In: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 708–717. DSN '07, IEEE Computer Society, Washington, DC, USA (2007), http://dx.doi.org/10.1109/DSN.2007.37 (Cited on pages 181 and 182)

[27] Boudali, H., Haverkort, B.R., Kuntz, M., Stoelinga, M.: Best of Three Worlds: Towards Sound Architectural Dependability Models. In: 8th International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS). pp. 45–49. CTIT Workshop Proceedings, University of Twente, CTIT, Enschede (September 2007), http://doc.utwente.nl/64339/ (Cited on page 2)

[28] Boudec, J., McDonald, D., Mundinger, J.: A Generic Mean Field Convergence Result for Systems of Interacting Objects. In: Proceedings of the Fourth International Conference on Quantitative Evaluation of Systems. pp. 3–18. QEST '07, IEEE Computer Society, Washington, DC, USA (2007), http://dx.doi.org/10.1109/QEST.2007.3 (Cited on page 200)

[29] Bouissou, M., Bon, J.L.: A new formalism that combines advantages of fault-trees and Markov models: Boolean logic driven Markov processes. Reliability Engineering & System Safety 82(2), 149–163 (2003), http://dx.doi.org/10.1016/S0951-8320(03)00143-1 (Cited on page 184)

[30] Bouissou, M., Dutuit, Y., Maillard, S.: Reliability Analysis of a Dynamic Phased Mission System: Comparison of Two Approaches. In: Modern Statistical and Mathematical Methods in Reliability, Quality, Reliability and Engineering Statistics, vol. 10, pp. 87–104. World Scientific Publishing Company (2005) (Cited on page 157)

[31] Bozzano, M., Cavada, R., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Olive, X.: Formal Verification and Validation of AADL Models.

In: Proceedings of Embedded Real Time Software and Systems Conference (2010), http://www.academia.edu/2742812/Formal_verification_and_validation_of_aadl_models (Cited on page 188)

[32] Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, Dependability and Performance Analysis of Extended AADL Models. The Computer Journal 54(5), 754–775 (2011), http://comjnl.oxfordjournals.org/content/54/5/754.abstract (Cited on page 188)

[33] Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M., Wimmer, R.: A Model Checker for AADL. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 6174, pp. 562–565. Springer (2010), http://dx.doi.org/10.1007/978-3-642-14295-6_48 (Cited on page 189)

[34] Bozzano, M., Cimatti, A., Lisagor, O., Mattarei, C., Mover, S., Roveri, M., Tonetta, S.: Symbolic Model Checking and Safety Assessment of Altarica models. In: Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS 2011). vol. 46. Electronic Communications of the EASST (2011), http://journal.ub.tu-berlin.de/eceasst/article/view/697 (Cited on page 190)

[35] Bozzano, M., Cimatti, A., Roveri, M., Katoen, J.P., Nguyen, V.Y., Noll, T.: Codesign of Dependable Systems: A Component-Based Modeling Language. In: Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign. pp. 121–130. MEMOCODE'09, IEEE Press, Piscataway, NJ, USA (2009), http://dl.acm.org/citation.cfm?id=1715759.1715776 (Cited on pages 182, 188, and 189)

[36] Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V., Noll, T., Roveri, M.: The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) Computer Safety, Reliability, and Security, Lecture Notes in Computer Science, vol. 5775, pp. 173–186. Springer (2009), http://dx.doi.org/10.1007/978-3-642-04468-7_15 (Cited on page 188)

[37] Brlek, S., Rauzy, A.: Synchronization of Constrained Transition Systems. In: Hong, H. (ed.) Proceedings of the First International Symposium on Parallel Symbolic Computation (PASCO'94). pp. 54–62. World Scientific Publishing, Linz, Austria (1994) (Cited on page 190)

[38] Bucklew, J.: Introduction to Rare Event Simulation. Springer Series in Statistics, Springer (2004) (Cited on pages 1 and 201)

[39] Cassez, F., Pagetti, C., Roux, O.: A Timed Extension for AltaRica. Fundamenta Informaticae 62(3-4), 291–332 (March 2004), http://dl.acm.org/citation.cfm?id=1227052.1227054 (Cited on page 190)

[40] Cavada, R., Cimatti, A., Franzén, A., Kalyanasundaram, K., Roveri, M., Shyamasundar, R.K.: Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In: Formal Methods in Computer Aided Design. pp. 69–76. FMCAD '07 (2007), http://dx.doi.org/10.1109/FAMCAD.2007.35 (Cited on page 189)

[41] Ciancone, A., Drago, M.L., Filieri, A., Grassi, V., Koziolek, H., Mirandola, R.: The KlaperSuite framework for model-driven reliability analysis of component-based systems. Software & Systems Modeling pp. 1–22 (2013), http://dx.doi.org/10.1007/s10270-013-0334-8 (Cited on page 186)

[42] Ciardo, G., Muppala, J., Trivedi, K.: SPNP: Stochastic Petri Net Package. In: Proceedings of the Third International Workshop on Petri Nets and Performance Models. pp. 142–151 (December 1989), http://dx.doi.org/10.1109/PNPM.1989.68548 (Cited on page 58)

[43] Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer (2002), http://dx.doi.org/10.1007/3-540-45657-0_29 (Cited on page 189)

[44] Cimatti, A., Franzén, A., Griggio, A., Kalyanasundaram, K., Roveri, M.: Tighter Integration of BDDs and SMT for Predicate Abstraction. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 1707–1712. DATE '10, European Design and Automation Association, Leuven, Belgium (2010) (Cited on pages 189 and 198)

[45] Clark, A., Gilmore, S., Hillston, J., Tribastone, M.: Stochastic Process Algebras. In: Bernardo, M., Hillston, J. (eds.) Formal Methods for Performance Evaluation, Lecture Notes in Computer Science, vol. 4486, pp. 132–179. Springer (2007), http://dx.doi.org/10.1007/978-3-540-72522-0_4 (Cited on page 1)

[46] Clark, G., Courtney, T., Daly, D., Deavours, D., Derisavi, S., Doyle, J.M., Sanders, W.H., Webster, P.: The Möbius Modeling Tool. In: Proceedings of the 9th international Workshop on Petri Nets and Performance Models (PNPM'01). pp. 241–250. PNPM '01, IEEE Computer Society, Washington, DC, USA (2001), http://dl.acm.org/citation.cfm?id=882474.883479 (Cited on page 187)

[47] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. Journal of the ACM 50(5), 752–794 (September 2003), http://dx.doi.org/10.1145/876638.876643 (Cited on page 189)

[48] Codd, E.F.: A Relational Model of Data for Large Shared Data Banks (Reprint). Communications of the ACM 26(1), 64–69 (1983), http://dx.doi.org/10.1145/357980.358007 (Cited on page 29)

[49] David, P., Idasiak, V., Kratz, F.: Automating the synthesis of AltaRica Data-Flow models from SysML. In: Bris, R., Guedes Soares, C., Martorell, S. (eds.) Proceedings of European Safety and Reliability Conference, ESREL 2009. European Safety and Reliability Association, Praha, Czech Republic (2009) (Cited on page 191)

[50] Design of Computer and Communication Systems Group (Inf 3) UniBw: LARES Website (2013), http://rocks.w3.rz.unibw-muenchen.de (Cited on page 203)

[51] Devlin, K.: The Joy of Sets: Fundamentals of Contemporary Set Theory. Springer, second edn. (1994) (Cited on page 27)

[52] Distefano, S., Puliafito, A.: Dynamic Reliability Block Diagrams VS Dynamic Fault Trees. In: Proceedings of the Annual Reliability and Maintainability Symposium (RAMS '07). pp. 71–76 (2007), http://dx.doi.org/10.1109/RAMS.2007.328095 (Cited on page 1)

[53] Donaldson, A.F., Miller, A., Parker, D.: Language-Level Symmetry Reduction for Probabilistic Model Checking. In: Proceedings of the Sixth International Conference on the Quantitative Evaluation of Systems. pp. 289–298 (September 2009), http://dx.doi.org/10.1109/QEST.2009.21 (Cited on page 200)

[54] D'Argenio, P.R., Hermanns, H., Katoen, J.P., Klaren, R.: MoDeST – A Modelling and Description Language for Stochastic Timed Systems. In: Alfaro, L., Gilmore, S. (eds.) Process Algebra and Probabilistic Methods. Performance Modelling and Verification, Lecture Notes in Computer Science, vol. 2165, pp. 87–104. Springer (2001), http://dx.doi.org/10.1007/3-540-44804-7_6 (Cited on pages 182 and 187)

[55] Eclipse Foundation: Epsilon (2013), http://www.eclipse.org/epsilon, [Online; accessed 10-September-2013] (Cited on pages 137 and 202)

[56] Eclipse Foundation: Graphiti – A Graphical Tooling Infrastructure (2013), http://www.eclipse.org/graphiti/, [Online; accessed 4-September-2013] (Cited on pages 137 and 202)

[57] Eclipse Foundation Inc.: Eclipse Modeling Framework Project (2013), http://www.eclipse.org/modeling/emf/, [Online; accessed 10-Oktober-2013] (Cited on pages 135 and 137)

[58] Eclipse Foundation Inc.: Xtext (2013), http://www.eclipse.org/Xtext, [Online; accessed 3-May-2013] (Cited on pages 30 and 134)

[59] Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis & Design Language (AADL): An Introduction. Tech. Rep. CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University (2006), www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA455842 (Cited on pages 182 and 188)

[60] Franks, G., Al-Omari, T., Woodside, M., Das, O., Derisavi, S.: Enhanced Modeling and Solution of Layered Queueing Networks. IEEE Transactions on Software Engineering 35(2), 148–161 (March 2009), http://dx.doi.org/10.1109/TSE.2008.74 (Cited on page 186)

[61] Friebert, J.: Entwicklung eines Editor- und Analyseumgebung für LARES-Modelle. Master's thesis, Universität der Bundeswehr München, Dept. of Computer Science 4 (in German) (2011) (Cited on page 137)

[62] Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A toolbox for the construction and analysis of distributed processes. International Journal on Software Tools for Technology Transfer 15(2), 89–107 (2013), http://dx.doi.org/10.1007/s10009-012-0244-z (Cited on page 183)

[63] Gauthier, J., Leduc, X., Rauzy, A.: Assessment of Large Automatically Generated Fault Trees by means of Binary Decision Diagrams. Journal of Risk and Reliability 221(2), 95–105 (2007) (Cited on page 190)

[64] Glabbeek, R.J.V., Smolka, S.A., Steffen, B.: Reactive, Generative, and Stratified Models of Probabilistic Processes. Information and Computation 121(1), 59–80 (1995), http://dx.doi.org/10.1006/inco.1995.1123 (Cited on page 7)

[65] Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer (1996), http://dx.doi.org/10.1007%2F3-540-60761-7 (Cited on page 200)

[66] Gouberman, A., Grand, C., Riedl, M., Siegle, M.: An IDE for the LARES Toolset. In: Fischbach, K., Krieger, U.R. (eds.) Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance, Lecture Notes in Computer Science, vol. 8376, pp. 240–254. Springer International Publishing (2014),

http://dx.doi.org/10.1007/978-3-319-05359-2_17 (Cited on pages 133, 136, and 137)

[67] Gouberman, A., Riedl, M., Schuster, J., Siegle, M.: A Modelling and Analysis Environment for LARES. In: Schmitt, J.B. (ed.) Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance, Lecture Notes in Computer Science, vol. 7201, pp. 244–248. Springer (2012), http://dx.doi.org/10.1007/978-3-642-28540-0_19 (Cited on page 133)

[68] Gouberman, A., Riedl, M., Schuster, J., Siegle, M., Walter, M.: LARES - A Novel Approach for Describing System Reconfigurability in Dependability Models of Fault-Tolerant Systems. In: Guedes Soares, C., Bri, R., Martorell, S. (eds.) Reliability, Risk, and Safety: Theory and Applications. Proceedings of the European Safety and Reliability Conference (ESREL '09), vol. 1, pp. 153–160. CRC Press (2009), http://dx.doi.org/10.1201/9780203859759.ch22 (Cited on pages 157, 158, and 161)

[69] Gouberman, A., Riedl, M., Siegle, M.: A Modular and Hierarchical Modelling Approach for Stochastic Control. In: Klement, E., Borutzky, W., Fahringer, T., Hamza, M., Uskov, V. (eds.) Proceedings of the 32nd IASTED International Conference on Modelling, Identification and Control (MIC '13). ACTA Press (2013), http://dx.doi.org/10.2316/P.2013.794-066 (Cited on pages 133, 134, 155, 194, and 196)

[70] Gouberman, A., Riedl, M., Siegle, M.: Transformation of LARES performability models to continuous-time Markov reward models. In: Proceedings of the 7th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECOS '13). eWiC, British Computer Society (2013) (Cited on pages 133, 134, 155, and 196)

[71] Grand, C.: Extension of a textual editor for the specification language LARES - Model transformation, validation and feature development. Master's thesis, Universität der Bundeswehr München, Dept. of Computer Science 4 (in German) (2013) (Cited on page 136)

[72] Graphical Editing Framework: (2013), http://www.eclipse.org/gef/, [Online; accessed 3-September-2013] (Cited on page 138)

[73] Graphical Modeling Project: (2013), http://www.eclipse.org/modeling/gmp/, [Online; accessed 10-September-2013] (Cited on page 137)

[74] Grassi, V., Mirandola, R., Randazzo, E., Sabetta, A.: KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability.

In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example, Lecture Notes in Computer Science, vol. 5153, pp. 327–356. Springer (2008), http://dx.doi.org/10.1007/978-3-540-85289-6_13 (Cited on pages 182 and 186)

[75] Grassi, V., Mirandola, R., Sabetta, A.: From Design to Analysis Models: A Kernel Language for Performance and Reliability Analysis of Component-based Systems. In: Proceedings of the 5th international workshop on Software and performance. pp. 25–36. WOSP '05, ACM, New York, NY, USA (2005), http://dx.doi.org/10.1145/1071021.1071024 (Cited on pages 185 and 186)

[76] Grassi, V., Mirandola, R., Sabetta, A.: A Model Transformation Approach for the Early Performance and Reliability Analysis of Component-Based Systems. In: Gorton, I., Heineman, G., Crnkovic, I., Schmidt, H., Stafford, J., Szyperski, C., Wallnau, K. (eds.) Component-Based Software Engineering, Lecture Notes in Computer Science, vol. 4063, pp. 270–284. Springer (2006), http://dx.doi.org/10.1007/11783565_19 (Cited on pages 182 and 186)

[77] Grassi, V., Mirandola, R., Sabetta, A.: A Model-Driven Approach to Performability Analysis of Dynamically Reconfigurable Component-Based Systems. In: Proceedings of the 6th international workshop on Software and performance. pp. 103–114. WOSP '07, ACM, New York, NY, USA (2007), http://dx.doi.org/10.1145/1216993.1217011 (Cited on page 186)

[78] Grumberg, O., Livne, S., Markovitch, S.: Learning to order BDD variables in verification. Journal of Artificial Intelligence Research 18, 83–116 (January 2003), http://dx.doi.org/10.1613/jair.1096 (Cited on page 215)

[79] Hahn, E., Hartmanns, A., Hermanns, H., Katoen, J.P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods in System Design pp. 1–42 (2012), http://dx.doi.org/10.1007/s10703-012-0167-z (Cited on page 187)

[80] Hansen, H., Timmer, M.: Why Confluence is More Powerful than Ample Sets in Probabilistic and Non-Probabilistic Branching Time. In: 10th Workshop on Quantitative Aspects of Programming Languages (QAPL 2012), Tallinn, Estonia. Istituto di Scienza e Tecnologie dell'Informazione, Pisa (April 2012), http://doc.utwente.nl/80456/ (Cited on page 200)

[81] Hartmanns, A.: MODEST – A unified language for quantitative models. In: Proceeding of the Forum on Specification and Design Languages. pp.

44–51. IEEE (2012), http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6336982 (Cited on page 187)

[82] Hartmanns, A., Hermanns, H.: A Modest Approach to Checking Probabilistic Timed Automata. In: Proceedings of the Sixth International Conference on the Quantitative Evaluation of Systems. pp. 187–196 (September 2009), http://dx.doi.org/10.1109/QEST.2009.41 (Cited on pages 187 and 197)

[83] Hermanns, H., Jansen, D., Usenko, Y.: From StoCharts to MoDeST: A comparative reliability analysis of train radio communications. In: Proceedings of the 5th international workshop on Software and performance, WOSP '05. pp. 13–23. ACM Press, New York, NY, USA (2005), http://doc.utwente.nl/54793/, hJU05 (Cited on page 187)

[84] Hermanns, H., Siegle, M.: Bisimulation Algorithms for Stochastic Process Algebras and Their BDD-Based Implementation. In: Katoen, J.P. (ed.) Formal Methods for Real-Time and Probabilistic Systems, Lecture Notes in Computer Science, vol. 1601, pp. 244–264. Springer (1999), http://dx.doi.org/10.1007/3-540-48778-6_15 (Cited on pages 120, 121, and 122)

[85] Hirel, C., Sahner, R., Zang, X., Trivedi, K.: Reliability and Performability Modeling Using SHARPE 2000. In: Haverkort, B., Bohnenkamp, H., Smith, C. (eds.) Computer Performance Evaluation.Modelling Techniques and Tools, Lecture Notes in Computer Science, vol. 1786, pp. 345–349. Springer (2000), http://dx.doi.org/10.1007/3-540-46429-8_28 (Cited on page 186)

[86] Hofmeister, C.R.: Dynamic Reconfiguration of Distributed Applications. Ph.D. thesis, University of Maryland, College Park, MD, USA (1993) (Cited on page 180)

[87] IEC: IEC 61508 Edition 2.0 – Functional Safety of Electrical / Electronic / Programmble Electronic Safety-Related Systems (April 2010) (Cited on page 18)

[88] ISO/IEC/(IEEE): ISO/IEC 42010 (IEEE Std) 1471-2000 : Systems and Software engineering - Recomended practice for architectural description of software-intensive systems (July 2007) (Cited on page 180)

[89] JFree: (2013), http://www.jfree.org/, [Online; accessed 4-September-2013] (Cited on page 139)

[90] Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The Ins and Outs of The Probabilistic Model Checker MRMC. Performance Evaluation 68(2), 90–104 (2011), http://dx.doi.org/10.1016/j.peva.2010.04.001, advances in Quantitative Evaluation of Systems, QEST 2009 (Cited on page 189)

[91] Kloul, L.: From DFTs to PEPA: A Model-to-Model Transformation. In: Bradley, J.T. (ed.) Computer Performance Engineering, Lecture Notes in Computer Science, vol. 5652, pp. 94–109. Springer (2009), http://dx.doi.org/10.1007/978-3-642-02924-0_8 (Cited on pages 181 and 182)

[92] Knuth, D.E.: Top-down syntax analysis. Acta Informatica 1(2), 79–110 (1971), http://dx.doi.org/10.1007/BF00289517 (Cited on page 30)

[93] Knuth, D.E.: Fundamental Algorithms, The Art of Computer Programming, vol. 1, Section 2.3, pp. 308–316. Addison-Wesley Professional, Reading, Massachusetts, third edn. (July 1997), TREES (Cited on page 29)

[94] Kuntz, M., Siegle, M., Werner, E.: Symbolic Performance and Dependability Evaluation with the Tool CASPA. In: Núñez, M., Maamar, Z., Pelayo, F.L., Pousttchi, K., Rubio, F. (eds.) Applying Formal Methods: Testing, Performance, and M/E-Commerce, Lecture Notes in Computer Science, vol. 3236, pp. 293–307. Springer (2004), http://dx.doi.org/10.1007/978-3-540-30233-9_22 (Cited on page 104)

[95] Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic Symbolic Model Checker. In: Field, T., Harrison, P., Bradley, J., Harder, U. (eds.) Computer Performance Evaluation: Modelling Techniques and Tools, Lecture Notes in Computer Science, vol. 2324, pp. 200–204. Springer (2002), http://dx.doi.org/10.1007/3-540-46029-2_13 (Cited on page 187)

[96] Lynch, N., Segala, R., Vaandrager, F.: Hybrid I/O Automata. Information and Computation 185(1), 105–157 (August 2003), http://dx.doi.org/10.1016/S0890-5401(03)00067-1 (Cited on page 198)

[97] Lückel, J., Grotstollen, H., Henke, M., Hestermeyer, T., Liu-Henke, X.: RailCab System: Engineering Aspects. In: Schiehlen, W. (ed.) Dynamical Analysis of Vehicle Systems, CISM International Centre for Mechanical Sciences, vol. 497, pp. 237–281. Springer Vienna (2009), http://dx.doi.org/10.1007/978-3-211-76666-8_6 (Cited on page 164)

[98] Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What Industry Needs from Architectural Languages: A Survey. IEEE Transactions on Software Engineering 39(6), 869–891 (2013), http://dx.doi.org/10.1109/TSE.2012.74 (Cited on page 180)

[99] Malek, M., Hoffmann, G.A., Milanovic, N., Brüning, S., Meyer, R., Milic, B.: Methoden und Werkzeuge zur Verfügbarkeitsermittlung. Informatik-Berichte 219,

Humboldt-Universität zu Berlin, Institut für Informatik (2007), http://edoc.hu-berlin.de/series/informatik-berichte/219/PDF/219.pdf (Cited on pages 179 and 181)

[100] Man, K.L., Schiffelers, R.R.H.: Formal Specification and Analysis of Hybrid Systems. Ph.D. thesis, Eindhoven University of Technology, Eindoven, Netherlands (2006) (Cited on page 201)

[101] Marciniak, C.: Erweiterung von LARES zur Modellierung von Entscheidungsprozessen. Master's thesis, Universität der Bundeswehr München, Dept. of Computer Science 4 (in German) (2012) (Cited on page 155)

[102] Martin, D., Burstein, M., Hobbs, E., Lassila, O., Mcdermott, D., Mcilraith, S., Narayanan, S., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., Sycara, K.: OWL-S: Semantic Markup for Web Services. Tech. rep., W3C (November 2004), http://www.w3.org/Submission/OWL-S/ (Cited on page 185)

[103] Meyer, T., Kessler, J., Sextro, W., Trachtler, A.: Increasing Intelligent Systems' Reliability by Using Reconfiguration. In: Proceedings of the Annual Reliability and Maintainability Symposium (RAMS '13). pp. 1–6 (2013), http://dx.doi.org/10.1109/RAMS.2013.6517636 (Cited on page 165)

[104] Meyer, T., Sondermann-Wölke, C., Sextro, W., Riedl, M., Gouberman, A., Siegle, M.: Bewertung der Zuverlässigkeit selbstoptimierender Systeme mit dem LARES-Framework. In: Gausemeier, J., Dumitrescu, R., Rammig, F., Schäfer, W., Trächtler, A. (eds.) 9. Paderborner Workshop Entwurf Mechatronischer Systeme. pp. 161–174. HNI-Verlagsschriftenreihe, Heinz-Nixdorf-Institut, Paderborn (2013) (Cited on pages 164 and 166)

[105] Norris, J.: Markov Chains. No. Nr. 2008 in Cambridge Series in Statistical and Probabilistic Mathematics, Cambridge University Press (1998) (Cited on page 1)

[106] Object Management Group: UML Profile for Schedulability, Performance, and Time Specification (2005), http://www.omg.org/spec/SPTP/1.1/ (Cited on page 186)

[107] Object Management Group: Meta Object Facility (MOF) Core Specification Version 2.0 (2006), http://www.omg.org/cgi-bin/doc?formal/2006-01-01 (Cited on pages 185 and 186)

[108] Object Management Group: UML 2.3 Superstructure (May 2010), http://www.omg.org/spec/UML/2.3 (Cited on page 182)

[109] Object Management Group: OMG Systems Modeling Language (OMG SysML$^{\text{TM}}$) (2012), www.omgsysml.org (Cited on page 3)

[110] Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A Comprehensive Step-by-step Guide. Artima Inc. (2008) (Cited on pages 140 and 141)

[111] Painter, R., Coppit, D.: Developing High-Level Reliability Languages Using A General Intermediate Domain. In: Proceedings of the Annual Reliability and Maintainability Symposium (RAMS '05). pp. 133–138 (2005), http://dx.doi.org/10.1109/RAMS.2005.1408351 (Cited on pages 181, 184, and 185)

[112] Petri, C.: Kommunikation mit Automaten. Ph.D. thesis, Institut für instrumentelle Mathematik, Bonn (1962) (Cited on page 1)

[113] Plotkin, G.D.: A Structural Approach to Operational Semantics. Journal of Logic and Algebraic Programming 60-61, 17–139 (2004), http://dx.doi.org/10.1016/j.jlap.2004.05.001 (Cited on page 94)

[114] Point, G.: AltaRica: Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement. Thèse de doctorat, LaBRI – Université Bordeaux I (January 2000) (Cited on pages 182 and 190)

[115] Point, G., Rauzy, A.: AltaRica – Constraint automata as a description language. Journal Européen des Systèmes Automatisés 33(8–9), 1033–1052 (1999) (Cited on pages 182 and 190)

[116] Pulungan, R.: Reduction of Acyclic Phase-Type Representations. Ph.D. thesis, Universität des Saarlandes, Saarbrücken, Germany (2009) (Cited on page 196)

[117] Rauzy, A.: Modes Automata and their Compilation into Fault Trees. Reliability Engineering and System Safety 78(1), 1–12 (2002), http://dx.doi.org/10.1016/S0951-8320(02)00042-X (Cited on page 190)

[118] Reijsbergen, D., de Boer, P.-T., Scheinhardt, W., Haverkort, W.: Rare event simulation for highly dependable systems with fast repairs. Performance Evaluation 69(7-8), 336–355 (July 2012), http://dx.doi.org/10.1016/j.peva.2011.11.004 (Cited on page 201)

[119] Riedl, M., Schuster, J., Siegle, M.: Recent Extensions to the Stochastic Process Algebra Tool CASPA. In: Proceedings of the 5th International Conference on Quantitative Evaluation of Systems (QEST '08). pp. 113–114 (2008), http://dx.doi.org/10.1109/QEST.2008.13 (Cited on pages 104 and 107)

[120] Riedl, M., Schuster, J., Siegle, M., Blum, M., Schiller, F.: Dependability Model Transformation – A Stochastic Process Algebra Semantics for ZuverSicht Models. In: Ale, B., Papazoglou, I., Zio, E. (eds.) Reliability, Risk and Safety: Back to the Future. Proceedings of the European Safety and Reliability Conference (ESREL '10). pp. 932–940. CRC Press London (2010) (Cited on page 108)

[121] Riedl, M., Siegle, M.: A LAnguage for REconfigurable dependable Systems: Semantics & Dependability Model Transformation. In: Proceedings of the 6th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECOS '12). pp. 78–89. eWiC, British Computer Society (August 2012) (Cited on page 10)

[122] Roy, P., Parker, D., Norman, G., de Alfaro, L.: Symbolic Magnifying Lens Abstraction in Markov Decision Processes. In: Quantitative Evaluation of Systems, 2008. QEST '08. Fifth International Conference on. pp. 103–112 (2008), `http://dx.doi.org/10.1109/QEST.2008.41` (Cited on page 200)

[123] Schuster, J.: Towards faster numerical solution of Continuous Time Markov Chains stored by symbolic data structures. Ph.D. thesis, Universität der Bundeswehr München (2012), `http://d-nb.info/102057920X` (Cited on pages 104, 106, and 158)

[124] Schuster, J., Siegle, M.: Dependability modelling with the stochastic process algebra tool CASPA. In: Proceedings of the First Workshop on DYnamic Aspects in DEpendability Models for Fault-Tolerant Systems (DYADEM-FTS '10). pp. 35–36. ACM, New York, NY, USA (2010), `http://dx.doi.org/10.1145/1772630.1772640` (Cited on pages 104 and 157)

[125] Schuster, J., Siegle, M.: Path-based calculation of MTTFF, MTTFR, and asymptotic unavailability with the stochastic process algebra tool CASPA. Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability 225(4), 399–406 (2011), `http://dx.doi.org/10.1177/1748006X11392286` (Cited on page 104)

[126] Siegle, M.: Behavior analysis of communication systems: Compositional modelling, compact representation and analysis of performability properties. Berichte aus der Informatik, Shaker (2002), `http://d-nb.info/965520692` (Cited on pages 120 and 200)

[127] Singh, D., Ibrahim, A.M., Yohanna, T., Singh, J.N.: A systematization of fundamentals of multisets. Lecturas Matematicas 29, 33–48 (2008) (Cited on page 27)

[128] Singh, D., Ibrahim, A.M., Yohanna, T., Singh, J.N.: Complementation in Multiset Theory. International Mathematical Forum 6(38), 1877–1884 (2011) (Cited on page 27)

[129] Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2002) (Cited on page 186)

[130] Society of Automotive Engineers (SAE): The SAE Architecture Analysis and Design Language (AADL) standard, `http://www.aadl.info/`, [Online; accessed 27-January-2014] (Cited on page 3)

[131] Sondermann-Wölke, C., Sextro, W.: Integration of Condition Monitoring in Self-optimizing Function Modules Applied to the Active Railway Guidance Module. International Journal on Advances in Intelligent Systems 3(1&2), 65–74 (2010) (Cited on page 164)

[132] Spivey, J.: The Z Notation: A Reference Manual. Prentice Hall International Series in Computer Science, Prentice Hall (1992), `http://spivey.oriel.ox.ac.uk/~mike/zrm/zrm.pdf` (Cited on page 184)

[133] Standard Widget Toolkit: (2013), `http://www.eclipse.org/swt/`, [Online; accessed 3-September-2013] (Cited on page 138)

[134] Torreborre, E.: $specs^2$ – Software Specifications for Scala (2013), `http://specs2.org`, [Online; accessed 13-December-2011] (Cited on page 155)

[135] Typesafe Inc.: The Akka Toolkit (2013), `http://akka.io/`, [Online; accessed 4-September-2013] (Cited on page 139)

[136] Villen-Altamirano, M., Villen-Altamirano, J.: Restart: A straightforward method for fast simulation of rare events. In: Simulation Conference Proceedings, 1994. Winter. pp. 282–289 (1994), `http://dx.doi.org/10.1109/WSC.1994.717150` (Cited on page 201)

[137] W3C: Scalable Vector Graphics (SVG) (2013), `http://www.w3.org/Graphics/SVG/`, [Online; accessed 3-September-2013] (Cited on page 139)

[138] Walter, M.: Simple Non-Markovian Models for Complex Repair and Maintenance Strategies with LARES+. In: Bérenguer, C., Grall, A., Soares, C.G. (eds.) Advances in Safety, Reliability and Risk Management. Proceedings of the European Safety and Reliability Conference (ESREL '11). pp. 962–969. CRC Press London (2011), `http://dx.doi.org/10.1201/b11433-135` (Cited on page 36)

[139] Walter, M., Siegle, M., Bode, A.: OpenSESAME - The Simple but Extensive, Structured Availability Modeling Environment. Reliability Engineering & System Safety 93(6), 857–873 (2008), http://dx.doi.org/10.1016/j.ress.2007.03.034 (Cited on pages 181 and 182)

[140] Wikipedia: Lares – Wikipedia, The Free Encyclopedia (2011), http://en.wikipedia.org/w/index.php?title=Lares&oldid=463655702, [Online; accessed 13-December-2011] (Cited on page 2)

[141] Zimmermann, A., Knoke, M., Huck, A., Hommel, G.: Towards version 4.0 of TimeNET. In: Proceedings of the 13th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB). pp. 473–476. VDE Verlag VDE Verlag (March 2006), http://ieeexplore.ieee.org/xpl/abstractAuthors.jsp?arnumber=5755407 (Cited on page 58)

Acronyms

AADL	Architecture Analysis & Design Language
AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
BDMP	Boolean logic Driven Markov Processes
CASPA	Composition and Analysis of SPA
CMS	Component Monitoring System
CPU	Central Processing Unit
CTL	Computation Tree Logic
CTMC	Continuous Time Markov Chain
DE	Deterministic Extension
DES	Discrete Event Simulation
DFG	Deutsche Forschungsgemeinschaft
DFT	Dynamic FT
DNF	Disjunctive Normal Form
EBNF	Extended Backus-Naur Form
EDA	Event Driven Automaton
eDSPN	Extended Deterministic and SPN
EMF	Eclipse Modeling Framework
EPL	Epsilon Pattern Language
EQN	Extended Queueing Network
ESA	European Space Agency
ESLTS	Extended Stochastic LTS
FA	Failure Automaton
FMEA	Failure Mode an Effects Analysis
FoD	Failure-on-Demand
FoR	Failure-on-Repair
FT	Fault Tree
FTN	Fault Tolerant Network
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IMC	Interactive MC
ISO	International Organization for Standardization
LARES	LAnguage for REconfigurable (dependable) Systems
LFA	LARES Flat Automata
LFIA	LARES Flat Interacting Automata
LGPL	GNU Lesser General Public License
IMC	Interactive Markov Chain
LQN	Layered Queueing Network
LTL	Linear Temporal Logic
LTS	Labelled Transition System
MC	Markov Chain
MDD	Model Driven Development
MDP	Markov Decision Process
mLQN	modified LQN
MOF	MetaObject Facility

ACRONYMS

MS	Monitoring System
MTBDD	Multi-Terminal BDD
MTTF	Mean-Time-To-Failure
NEDA	Network of EDAs
OOP	Object Oriented Programming
OWL	Web Ontology Language
PAC	Process Algebra Composition
PACT	Process Algebra Composition Tuple
PAT	Process Algebra Tuple
PMS	Phased Mission System
PN	Petri Net
PTL	Probabilistic Timed Logic
QoS	Quality of Service
QVT	Query View Transformation
RBD	Reliability Block Diagram
RE	Reward Extension
RG	Reachability Graph
SAT	Satisfiability
SLTS	Stochastic LTS
SMC	Statistical Model Checking
SMR	semi-Markov reward (model)
SMT	Satisfiability Modulo Theory
SOS	Structural Operational Semantic
SPA	Stochastic Process Algebra
SPE	Software Performance Engineering
SPN	Stochastic PN
SVG	Scalable Vector Graphics
SWT	Standard Widget Toolkit
SysML	System Modelling Language
TS	Transition System
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	Extensible Markup Language