Preserving Digital Objects

A Constraint-Based Approach for the Automated Application to Transformation Processes

Thomas Triebsees

Dissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften

> *Digital documents last forever -*
> *or five years, whichever comes first.*
>
> *Jeff Rothenberg: Avoiding Technological Quicksand, 1999*

1. Berichterstatter: Prof. Dr. Uwe M. Borghoff
2. Berichterstatter: Prof. Dr. Gunther Schmidt

Fakultät für Informatik
Institut für Softwaretechnologie

Neubiberg, den 24. April 2008

Contents

Abstract

Rapid technology evolution lets Digital Archives (DA) face great challenges in *preserving* the contents of *digital* objects. A standard approach to preservation is to *migrate* digital objects to new technologies periodically. There, object representations may change whereas their contents must not. In large-scale scenarios *automated quality assurance* is a major concern: Did a given migration process preserve all relevant object properties? However, automation is often hindered as preservation requirements are expressed *informally*. In these cases, quality assurance is often hand-crafted, which is time-consuming, expensive, and error-prone.

We introduce a framework that is designed to support *automation* of migration and quality assurance processes in digital archiving. In particular, we *express* semantic preservation requirements formally; automated routines test migration processes for adherence to them. Theoretic well-foundedness and smooth integration into internal workflows of DAs have been important design goals.

A customizable, state-based archival context enables workflow integration. Here, we presuppose little system knowledge only: Objects must be uniquely identifiable. Users can integrate domain-specific *object types* and functionality. Well-defined state changes capture the effects of migration processes. There, our system ensures object immutability based on a formal notion of object contents.

Semantic requirements of the form "When transforming objects $o_1, ..., o_n$, preserve property ϕ" *constrain* migration processes. This *preservation language* bases on full first order logic. Object properties are captured by so-called *concepts*. Preservation requirements refer to concepts *by name* so that implementation details are hidden. This keeps specifications *readable* and less prone to changing implementations.

Our generic *notion of preservation* relates (1) source and target objects, (2) object histories, and (3) concepts. A concept is *preserved* if the target objects are new versions of the source objects and the concept equally holds for the source and target objects. There, we permit different concept implementations for the source and target objects — we support *content migration*.

When migrations are executed, our system *traces* changes to digital objects and *reports* constraint violations automatically. There, object traces derive from (iterated) object transformations. Concept *interfaces* allow for partial, thus, efficient tracing here. Reports relate concrete source and target objects to violated requirements. This facilitates adequate and customized reactions.

Due to a coherently formal underpinning, our methods satisfy a high degree of "trustworthiness". A case study in the field of website migration shows that our methods are applicable, beneficial, and scale to a relevant problem size. In the case study we also demonstrate formal model construction facilities of our framework. We have developed a general approach to integrating graph-based queries and apply these methods to automated URL construction. Runtime measurements show acceptable performance when using our prototype implementation.

Acknowledgements

I would like to thank everyone who supported me while I was writing this thesis. Some people, however, should receive special mention. First, and foremost, I thank my supervisor Uwe M. Borghoff for his continuous advice and for "pushing" me towards early publications. Reviewers have provided most valuable comments on our conference and journal submissions. I am convinced that this has increased relevance and quality of our work. Second, I thank my advisor Gunther Schmidt. He has provided most valuable comments on the formal parts of this thesis. Especially in the early phases he made me "think globally" instead of delving too deeply into theoretical aspects. Furthermore, my thanks go to the staff of the Institute for Software Technology. They have provided a friendly, lively, and creative research climate. In particular, I am grateful to Lothar Schmitz, Peter Rödig, and Steffen Mazanek for many fruitful discussions and for proof-reading parts of this thesis. Also, I thank all other members of the Department for Computer Science. Many interesting discussions and joint conference visits have provided variety and have expanded my scope to all the other interesting fields of Computer Science. Finally, I thank my family and my friends for their patience, their support, and for continuously reminding me that there is something beyond this work; to Inga Schmundt I am particularly grateful for proof-reading parts of this thesis.

Part I

Introduction and Informal Survey

Chapter 1

Introduction and Motivation

Long term preservation of non-digital objects has been a well-known issue for a long time. Museums, libraries and archives preserve artifacts like books, paintings, sculptures or other works of art that are part of our cultural and intellectual heritage.

Since the emergence and world-wide deployment of Personal Computers, however, we have recognized an ever-increasing growth of *digital* material [BRSS06]. The advantages of digital deployment are obvious. Digital objects can be processed automatically, they can be accessed almost barrier-free over the World Wide Web, and they support automated full-text search as well as "perfect" copies. This, e.g., led to mass-digitization of books [Goo07]. In addition, more and more artifacts are "born digital". In this respect, online-publishing of research results, the "paperless office", and fully digital document workflows have been propagated and implemented for quite a while.

Therefore, long term preservation of digital material has become an urgent issue over the past decades. It turned out that Digital Archives (DA) with long-term focus face great challenges [Tas96, Con02]. Their aim is to *preserve* digital objects over long (i.e., essentially undefined) periods of time. However, there is an important difference between non-digital and digital objects. Any non-digital object can be perceived through the five human senses; no digital object can be perceived without hardware and software environments. For this reason, digital preservation comprises keeping digital objects readable, understandable, and processable [Con02, BRSS06]. In that, the rapid technological evolution of hardware and software environments frequently causes DAs to transfer digital objects from obsolete technologies to newer ones.

The literature describes and examines two major approaches for that purpose — emulation and migration. Emulation does not affect the digital objects directly. It is rather used to emulate older hardware environments on newer ones. In this way, the original objects can be read on newer hardware technologies while using old software. In this thesis, however, we will be concerned with migration. This approach is used to directly transfer the original object to the new technology and may well induce changes to the object itself. Although parts of the objects change, one still wishes to preserve the information they carry. File format transformation is a typical scenario. Since the emergence of the XML-technology [Wor04b] and the resulting diversity of domain-related formats, migration has become part of day-to-day business in automated knowledge exchange. The information content of the exchanged objects, however, must not change.

Database migration, file format transformations or migration and deployment of software products on different platforms are good examples to show the general complexity of migration tasks. Plenty of interrelated digital objects have to be migrated and several semantic relationships have to be preserved. It is easy to comprehend that migration, if automated and planned in an ad-hoc manner, can lead to unintended effects and information loss. Hence, quality assurance becomes a major concern. Often enough, migration (and already ingesting objects into an archive can be considered to be a migration), is either done hand-crafted by human actors [GW05] or human reviewers have to go through objects "by hand" that are themselves produced semi-automatically, only. This is time-consuming, expensive, and still error-prone. In addition, semantic relationships and preservation requirements being expressed informally often prevent automatic checks.

By the end of 2005, leading researchers in the field of digital preservation met in Warwick, UK, and defined the research agenda for the next decade. In their final report [GW05], they identify the necessity to "...develop data description tools and associated generic migration applications to facilitate automation..." and to "...develop code generation tools for automatically creating software for format migration..." as part of the research agenda for the next five years. Since migration is the most widely practiced preservation strategy and still lacks sufficient automated support [Dig01, GW05], it is high time to integrate automation techniques and digital archive migration.

In this thesis, we propose a general framework that meets parts of these challenges and supports archivists in planning and running migrations. The major objective is to increase the level of automation and to improve the quality of executed migrations. In the next sections we will introduce important general conditions that drove the development of our approach. In particular, we will give reasons why and explain how we restrict the application domain. We will present the key ideas of our approach and motivate it with the application domain. Additionally, we will summarize the objectives of this thesis and provide the reader with an outline and a reading guide.

1.1 General Conditions of the Application Domain

In order to illustrate the environment in which this thesis is settled, Fig. 1.1 depicts a simplified schematic view on DAs. It can be seen as a reference structure identifying the basic components and actors. First, DAs have an *organizational* component, in which policies and practices of the corresponding archive are fixed. They serve as overall guide for the preservation of the *digital objects* that are brought into the archive by *producers* (today) and are potentially used by *consumers* somewhen in the future. Preservation over a long period of time is a complex task and faces several challenges like technical decay of data storage media or changing hardware and software technologies. Therefore, digital objects usually undergo several *processes*. These processes are supported by the *technical infrastructure* of the archive and possibly by external *vendors and suppliers*.

This — deliberately vaguely formulated — description, however, leaves open many questions. What are digital objects? What kind of processes run inside an archive? What does preservation mean? How does migration come into play?

The Reference Model for Open Archival Information Systems (OAIS, [Con02]) serves

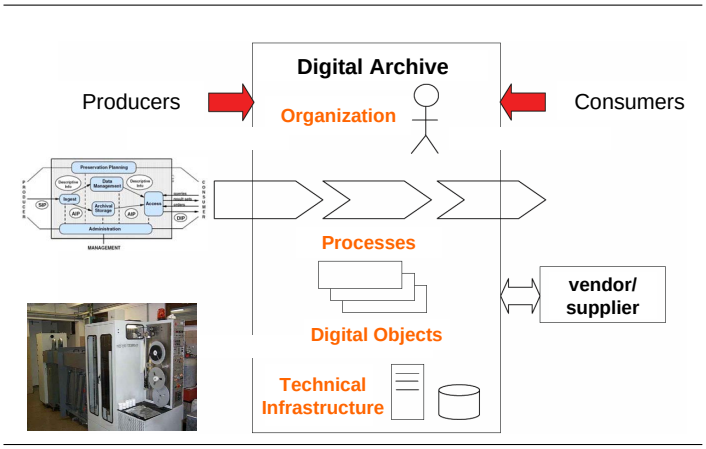


Figure 1.1: Simplified view on a Digital Archive

as the entry point to get first answers to these questions. It directly addresses the problem of preserving digital material over a long period of time and is widely accepted nowadays as a reference model. The archive structure shown in Fig. 1.1 is drawn from the descriptions in [Con02]. In particular, the OAIS reference model

- introduces the basic terminology,
- provides an abstract information model for digital objects,
- includes an organizational model for DAs,
- defines a basic functional model for processes that run inside DAs, and
- describes concrete preservation methods like migration and emulation.

All these aspects bear inside general conditions that drove the development of the approach presented in this thesis. Therefore, we will deal with them separately in the following subsections and describe how they influenced the development of the framework and restrict the application domain. In that we will introduce the basic terminology used throughout this thesis and briefly answer the questions raised above.

1.1.1 Information and Object Model

An appropriate information and object model is the basis for the introduction of further concepts. Since the OAIS model is a *reference model*, it structurally covers a broad variety of digital objects(see Fig. 1.2). The left-hand part of Fig. 1.2 shows the strict distinction between *Data Objects* and *Information Objects*. Data objects represent the pure, unrepresented data and consist of bit streams. This is indicated by a floppy disk icon and modeled by the aggregation relation in the right-hand part of Fig. 1.2. Using the *Representation Information* that is attached to a data object, the representation process yields an information object. In this example, we — at least — need a browser (program icon) and a style sheet for being able to render the data object as a web page. This web page can be considered to be the information object that represents the bit stream on the floppy disk. The information object is the real intellectual property which is to be preserved. When we speak of *digital objects* in the sequel, we refer

to the information object. Notice that representation information can contain other representation information, which yields *representation networks*.

The generality of this information model shows the potential complexity and diversity of digital objects. To mention some, today's DAs administrate audio and video files, databases, images, websites, office documents like presentations, spread sheets, or text documents, electronic mails etc. All these different kinds of digital objects require different preservation policies [Dig01]. Emulation, for example, seems to be the most promising approach for the preservation of complex databases and multi media files [Dig01]. In particular, if legal aspects are involved, it can otherwise be difficult (if not impossible) to guarantee integrity of these objects. Additionally, the content of multimedia files is often compressed or scrambled, which prevents direct access.

In contrast, this is no issue for websites. They are usually deployed using text-based file formats. However, embedded scripts, or referenced external resources (like databases) complicate preservation over the long term. Moreover, websites are usually highly interrelated such that link-consistency is a problem.

These observations demand for a restriction of the class of objects that can be handled by our approach. This will be done in Sect. 1.2. We shall, however, see that we still cover a broad class of digital objects.

1.1.2 Organizational and Functional Model

In order to maintain availability and understandability of the stored digital objects, several archiving activities necessitate data re-processing and data exchange between archival components. For this purpose, the OAIS reference model identifies five *Functional Entities*: (1) *Preservation Planning*, (2) *Administration*, (3) *Ingest*, (4) *Data Management*, (5) *Archival Storage*, and (6) *Access*. Their interrelations are depicted in Fig. 1.3.

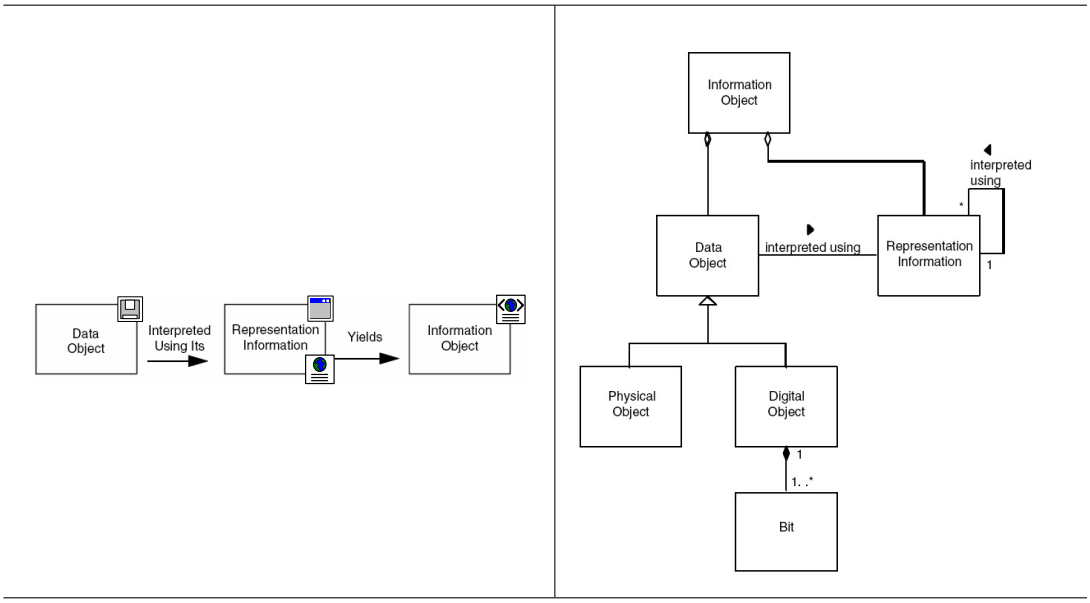


Figure 1.2: Difference between data and information ([Con02])

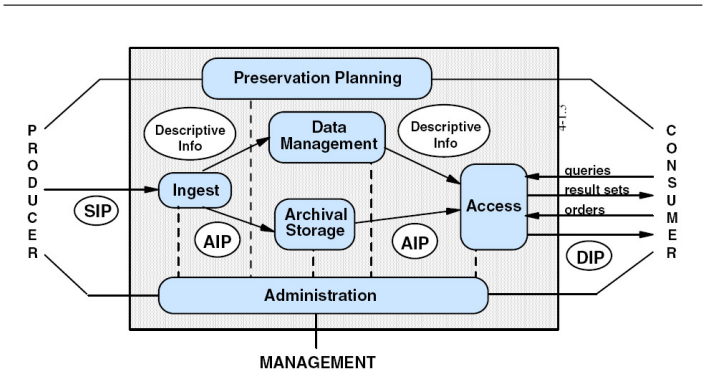


Figure 1.3: Functional Entities in the OAIS reference model ([Con02])

These entities serve to

- define general policies, practices, procedures, and preservation strategies (*Preservation Planning*),
- control adherence to these regulations on a day-to-day basis (*Administration*),
- provide catalogs and inventories on what may be retrieved from the archive including algorithms and procedures that can be run on retrieved data (*Data Management*),
- provide services and functions for the storage, maintenance and retrieval of digital objects (*Archival Storage*),
- provide services and functions to accept submissions from producers and prepare the contents for storage and management (*Ingest*), and
- provide services and functions that support consumers in determining the existence, description, location and availability of information stored in the OAIS, and allowing consumers to request and receive information products (*Access*), respectively [Con02].

Preservation Planning and *Administration* are two central organizational entities. They are particularly important when it is up to a decision whether to migrate digital objects. According to [Con02], "*Preservation Planning* [...] develops detailed Migration plans, software prototypes and test plans to enable implementation of *Administration* migration goals". *Administration* then "... also provides system engineering functions to monitor and improve archive operations, and to inventory, report on, and migrate/update the contents of the archive...". The approach developed in this thesis is designed to *support* these organizational entities. In particular, our framework is expected to

- support *Preservation Planning* in *expressing* their preservation requirements and
- provide *automated* support for *Administration* in checking whether the migration results meet these requirements.

Therefore, the requirements must be expressible in a sufficiently formal way and our framework should smoothly integrate into internal migration processes.

Fig. 1.3 also shows the several types of *information packages* that have been identified in [Con02]. We do not go into detail with it but want to mention that objects already

undergo non-trivial processes before they are stored somewhere permanently or issued to the consumer. On ingest, e.g., incoming objects are packaged into a *Submission Information Package* (SIP). Then they are adapted to an internal storage model called *Archival Information Package* (AIP), which usually includes addition of meta data and re-packaging. When making these objects available to the public or exchanging them between different archives, the dissemination process can include transformations to an exchange format called *Dissemination Information Package* (DIP). In Sect. 1.1.3 we shall see that this already meets the definition of *migration* of [Con02]. These internal processes exhibit three important facts:

(1) Different kinds of meta data are usually attached to digital objects. These meta data serve different purposes like describing the provenance or context of the digital objects. In particular context information can comprise semantic meta data that is, e.g., formulated using *Dublin Core* ([DCMID08]), the *Resource Description Framework* (RDF, [Wor99]), or *Topic Maps* ([Top01]). In any case, relationships between objects and their meta data are important and must not get lost. Hence, semantic object interrelations will play a central role in this thesis.

(2) Digital objects follow life cycles from the producer to the consumer. Whenever migrations produce unwanted results, reversibility becomes an important factor. The easiest way to achieve this is to retain the original objects and store their *history*. Hence, the necessity to *trace object histories* has guided the development of our approach significantly.

(3) As archives usually maintain large bodies of digital objects, *automation* is crucial. Hence, one may ask for *trustworthiness*. More specifically: Can one really trust in the fact that the consumer somewhen in the future receives the same (or seemingly the same) digital object that has been archived by the producer today? Especially if legal aspects are involved, this plays a key role. Suppose the paperwork of a specific court trail has been archived in electronic form and is needed for another trail in the future. Clearly, the consumer then expects to receive a copy the content of which "equals" the original. Things get even more serious for DAs if clients can claim damage recovery. In line with the conclusions from above, we interpret the demand for trustworthiness as follows: Preservation requirements must be expressible and adherence to them must be *traceable* in a sufficiently formal way. In this way, we can recapitulate an objects history and conclude what has been and what has not been preserved up to the "current version" of this object.

1.1.3 Migration and Preservation

As this thesis aims at a framework for the automated support of migration processes, we need to clarify the notion "migration" in the context of DAs. The OAIS reference model identifies four forms of migration.

(1) The bitwise copying of stored AIPs to another instance of the same storage media type is called *Refreshment*. A bitwise clone of a CD is a refreshment. This is well-handled in practice.

(2) *Replication* is an extended variant of refreshment, where the storage media type may change. Copying a file to another location, e.g., is a replication provided that the bit sequence is identical.

(3) A *Repackaging* affects the packaging information that is needed to address and access the digital object. The transfer from a CD to a DVD is a typical scenario. Assume a digital object is stored on a CD that implements the ISO9660 file system. Furthermore, let the DVD implement the UDF file system. Copying the file-based structure and content of the digital object to the DVD yields a repackaging because the DVD implements a different file system.

(4) *Transformation* is the most challenging variant of migration [Con02, Dig01]. It directly affects the content of the underlying digital object and may include the other variants as well. File format transformation is a typical scenario; [Con02] describes the change of file encodings from ASCII to UTF-8 as another example.

Object transformations introduce a variety of complex issues. We explain some of them in the following using a file format transformation from Microsoft Word to PDF. First, the original digital object may be copyrighted. Automatically, the question arises whether the archive has appropriate rights to transform (i.e., potentially adulterate) this object's content. Similar questions come up when the original document is digitally signed. Even if the original document is not copyrighted, integrity is an issue. Maintaining integrity means preventing the original object from being altered or destroyed in an unauthorized way [Con02]. Second, the producer of the object may doubt originality of the result. Is the PDF still understood in the same way as the original document? Already small layout changes (unintended page breaks etc.) can affect the answer to this question. Third, Microsoft Word is a proprietary format. This hampers or even prevents access to the internal structure of the original document and complicates the implementation of appropriate transformations.

Throughout this thesis we will abstract from these aspects. Concerning Digital Rights Management (DRM), we refer the interested reader to [SL05]. Furthermore, we assume that the archives' security policies assure integrity of all hosted objects. This is in wide areas a matter of the technical environment (access to permanent storages etc.) and is beyond the scope of this thesis. The question of originality of the migration result has already been answered. Although digital migration has a "...focus on the preservation of the full information content..." and "...a perspective that the new archival implementation of the information is a replacement for the old..." ([Con02]), one can say that the transformation result is only a new *version* of the original. Also, we neglect the aspect of proprietary file formats and assume that tools are available for accessing the digital objects and exploring their internal structure. This is virtually no limitation since most archives accept non-proprietary formats only, and archive policies mostly require archiving formats to be open standards.

In the sequel we will use the notions *migration* and *transformation* as synonyms in order to emphasize that migration processes potentially change object content. In the last subsection we have already raised the question of "trustworthiness". More specifically: Can we rely on the fact that the new version of the original object represents the original information content? In order to give a reliable answer to this question, we will have to clarify the notions *information* and *preservation* in more detail and rate the

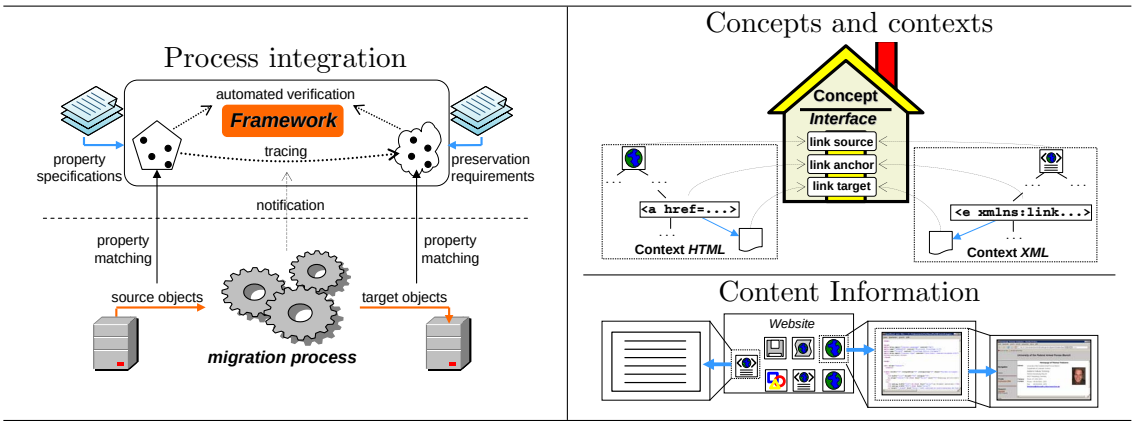


Figure 1.4: Overview of the approach of this thesis

quality of migration processes w.r.t. the information they preserve.[1]

Among others, the following three facts about information and preservation are widely accepted:

(1) The information does *not* comprise the bit stream only; there can be different views on what the information of a digital object is [Con02, Dig01, PRE05]. As an example, take a website that consists of a source directory and a collection of resources. These entities as well as the fact that they belong to this website can be considered (part of) this website's information. So can the content and "look and feel" of the resources when they are rendered by a browser. Therefore, our approach will have to support the *definition* of information pieces w.r.t. digital objects.

(2) It is hardly feasible to preserve *all* information that is carried by digital objects [Con02, Whe01, Dig01]. Our approach will have to facilitate the specification of preservation requirements directly.

(3) Applications of format transformation show that information or the process of retrieving information can be implemented differently for the source and target objects. These different implementations will have to be considered with our notion of preservation.

In the next section we survey the key ideas of our approach.

1.2 The Approach in this Thesis

In the left-hand part Fig. 1.4 illustrates how our approach conceptually integrates into migration processes that run inside DAs. Generally, our framework is designed to build on top of implemented migration processes while causing only a minimal influence to the processes themselves. In order to check whether a migration process meets given preservation requirements, our framework needs two formal inputs: (1) a specification of relevant semantic object properties and object relationships and (2) the preservation requirements for the migration process. As soon as a migration process starts, our

[1] w.r.t. = with respect to

framework matches the known properties to the source objects. Whenever the migration process executes "relevant" operations such as transforming an object or creating an object, we require the process to communicate this operation to our framework. In this way, we can *trace* the source objects, i.e., relate them directly to their respective migration targets. As soon as the migration process is completed, our framework checks the preservation requirements w.r.t. the source and the target objects. Hence, the ability to trace semantic object properties through running migration processes and along different versions of digital objects is a key ingredient of our framework. We implement this facility using *Abstract State Machines* (ASM, [BS03]). In this way, we receive a state-based operational system semantics for DAs which is both sound and natural. It allows us to capture effects of migration processes on the whole archive. Whenever a migration process communicates the events from above, our framework executes an appropriate "formal state change" as well. Preservation requirements are then checked w.r.t. a source state (starting point of the migration) and the final state (finishing point of the migration). Apart from that, this state-based view provides a natural semantics for rollbacks or "undos". This is an important property for practical applications; financial or other resource insufficiencies can cause migration processes to be interrupted and switched back to the last "sensible" archival state.

Apart from this conceptual overview, some technical questions remain open. We have already mentioned that information or the process of retrieving information can be implemented differently for the source and target objects. We cover this by so-called *concepts* and *contexts*. The upper right-hand part of Fig. 1.4 depicts a schematic overview of this approach. A concept groups different implementations of a semantic property of one or more (interrelated) objects. For this purpose, concepts include an *interface*. The semantic property can then be implemented in different contexts, where these implementations must adhere to the interface. This is a well-known technique in programming languages. The example in Fig. 1.4 shows an html document that contains a link to another resource. This property can be expressed as follows: The html document contains a link anchor, the reference attribute of which points to the respective resource. XML links, for example, are implemented similarly using a link source, a link anchor, and a link target. Hence, the concept's interface in Fig. 1.4 comprises these three entities. On an appropriately low level of abstraction, the implementations of html links and XML links differ (cf. the different implementations of the link anchors in Fig. 1.4). These different implementations are supported by our approach. Since the concept's interface is common to *all* its implementations, it serves as tracing point for the preservation of specified properties. We use history traces together with the state-based semantics in order to report violations of preservation requirements and their reasons.

The explanations so far already indicate our view on *information* and *preservation*. Throughout this thesis we will use a technical notion of information. It is sketchily depicted in the lower right-hand part of Fig. 1.4. We consider all content-based properties to be part of an object's information as well as everything that can be "derived" therefrom. In Fig. 1.4 we illustrate this by a website that consists of a collection of resource files. These entities as well as their textual content (indicated by a derivation arrow for some icons) can be part of the information carried by this website. So can the "look and feel" of an html file when rendered by a browser (outermost right-hand

derivation). In that, we distinguish *content information* and (semantic) *relationships*, which is in line with the most relevant literature [PRE05, Con02]. Content information (like the textual content of a file) usually has a certain *value* and can be derived directly from an object's representation. Other content information (like the first line of the textual content) can possibly be derived from it. In contrast, semantic relationships (like linking) usually hold between several objects. The value of a semantic relationship is restricted to "does hold" or "does not hold". No further content information is derivable. As already mentioned, this notion of information is a technical one. It does not serve to capture the "real meaning" or "intended meaning" of a document. It rather deals with the technical properties that are considered to be relevant for *extracting* the meaning of a digital object. Neither does the pure textual content of a website (content information) nor the fact that all links are working (semantic relationship) reflect the "meaning" of a website. Yet the meaning of a website cannot be extracted without them. Therefore, preserving these properties is considered to be vital in web archiving [PRE05].

Altogether, our notion of information has an important property. It is *reasonably formalizable* and can be *specified* w.r.t. a digital object. We argue that sufficient formalization is a prerequisite for *trustworthy automation*. The facility to identify, extract, and process only parts of an objects information helps in handling complexity. We can focus on "critical" aspects (*significant properties*) and neglect properties the preservation of which is not worth formal treatment due to their simplicity or unimportance. Link consistency may be considered to be critical since the meaning of a website may get lost if links do not work. On the contrary, directory and file structures "in the background" might be unimportant.

Using this view on *information*, preservation can — roughly speaking — be understood as follows:

> When an object is migrated and a piece of information can be derived from this object's properties, this information piece is preserved, if it can be derived from the transformation result as well.

This is in line with our approach using concepts and contexts from above. In Chapter 2 we will discuss this notion of preservation in more detail and illustrate its different facets with an example. At this point, however, we already mention that preservation requirements are expressed w.r.t. specified concepts. In particular, concepts are addressed by their name, which is globally unique. Using concepts, we hide implementation details, and raise specifications to an abstract level. We claim that this *preservation language* is more easily understood by archivists than the technical implementations. Also, these specifications are not prone to changing implementations.

We conclude this section by listing restrictions of our approach. Most of them have already been motivated by the preceding explanations.

- *Digital objects:* The content and structure of digital objects is accessible and processable in an automated way by the DA. Objects do not contain dynamic content. In particular, our approach does not cover software, video, or audio apart from their fixed manifestation (e.g., as a file). The DA has all permissions to access and process the stored digital objects; our approach does not innately cover any issues related to digital rights management (DRM).

- *Information and preservation:* Information content of and relationships between digital objects are reasonably formalizable; we implement a "technical" notion of information. Preservation is restricted to the preservation of formalized aspects. In particular, we do not capture aspects like human cognition.
- *Digital Archives:* Our approach does not cover any organizational or technical issues apart from articulating and verifying preservation requirements concerning the preservation of information content of digital objects during migration processes. We do not model complex internal workflows and processes, preservation approaches other than migration, or any costs arising from a migration. Hence, our approach cannot be used for strategic planning in terms of finding the best preservation strategy or estimating the *potential* effects of a migration; we presume the decision pro migration has been made.

Yet these limitations still let our approach cover a broad variety of digital objects and preservation scenarios. With permission of the Universität der Bundeswehr München, preliminary results of this thesis have been published in [TBS05, TB06, TB07a, TB07c, TB07b, Tri07]. In [TB07c, Tri07] we have evaluated our method in the context of software specifications. Therefore, we argue that our approach can be used in application domains other than digital archiving as well.

1.3 Objectives

The overall objective of this thesis is as follows.

> The results of this thesis contribute to a better automation of quality assurance for migration processes in digital archiving. Basing on suitable formal notions of information and preservation, our method allows to *express* preservation requirements formally and *evaluate* adherence to them in an automated way. The formal underpinning of our approach results in a high degree of trustworthiness. Practicability, well-foundedness, the ability to *pinpoint* violations of preservation requirements and smooth integration into internal workflows of DAs have been important design goals for our method.

Sub-ordinate objectives are:

- We define a customizable, state-based environment that captures digital objects, object contents, object relationships, and migration processes.
- We model formal object properties that are subject to preservation and can be implemented in different ways.
- We introduce a preservation language that can be used to specify the preservation of object properties in a formal way. Along with a formal notion of preservation, the semantics includes history traces for objects and, thus, enables automated evaluation of preservation requirements.
- We define the syntax and the semantics of a programming language for migration processes. The semantics integrates into our state-based environment. It, thus, facilitates automated and formal quality assurance; migration processes that are

implemented in our programming language can be checked w.r.t. adherence to given preservation requirements in an automated way.

- We prove practicability and usefulness of our methods by a case study in the field of website migration. Runtime measurements using our prototype implementation underline that our approach scales to a relevant problem size.

1.4 Outline and Reading Guide

We organize this thesis into five parts. Apart from this introduction, Part I contains an expanded informal overview of our approach (Chap. 2). There, we introduce the running example for the thesis (a website transformation) and apply our approach to automating quality assurance on an informal level.

In Part II, we formalize our approach. We start by introducing a formal state-based digital archive in Chap. 3. This includes formal object types, a formal notion of object content, and basic state changes. Migrations are defined as sequences of these state changes. In Chap. 4 we define syntax and semantics of *functional and non-functional concepts*. There, we distinguish a *static* and a *dynamic* semantics. Concepts are evaluated in a dynamic environment where objects may not exist. Dynamic concept semantics is strict and evaluates to "undefined" if concept implementations refer to non-existing objects. We close part two by introducing our preservation language (Chap. 5). We define object traces and a preservation predicate formally. Then we introduce basic *functional* and *non-functional preservation constraints*; they refer to functional and non-functional concepts, respectively. Existential and universal trace quantifiers are used to handle branching histories. Finally, *preservation formulas* add facilities to select object collections and to apply preservation constraints to all their members.

In order to improve usability (Part III), we provide a functional programming language for implementing migration processes (Chap. 6). We start by introducing *state change operations* that implement the state changes of Chap. 3. These operations are then integrated into a functional language. The language semantics generates formal migration sequences and, thus, smoothly integrates into our dynamic environment. We conclude Part III by integrating regular graph query languages. In general, expressing graph queries tends to be laborious as we use FOPL. We, however, define a specification scheme for query semantics and a method for combining query syntax (context-free grammars) to query semantics appropriately. This yields so-called *dominated product automata*, which can be used to evaluate and construct graph-based queries automatically. We apply these techniques to automated URL construction.

In Part IV we present the full case study. There, we evaluate our methods using the website migration example of Chap. 2. This comprises datatype specifications (Chap. 9), implementing the transformation process (Chap. 10), concept specifications (Chap. 11), and expressing preservation requirements (Chap. 12). Chap. 12 includes performance measurements for our prototype system. Costs and benefits of our formal quality assurance approach are discussed in Chap. 13.

Part V concludes this thesis. Related work is discussed in Chap. 14. In Chap. 15 we summarize our contributions. In particular, we point up how we have met our initial objectives and point out directions for future research.

Reading Guide

The following reading order provides a significant overview of our contributions but skips all technical parts. We suggest if for a *first reading* and for those who have *no technical background*.

Introduction:
Chap. 1 (Introduction)
Chap. 2 (Informal Survey)
Case study:
Prolog of Chap. 8 (Case Study)
Sect. 12.2 (Evaluating Runtime Costs)
Chap. 13 (Summary — Costs and Benefits)
Conclusions:
Sect. 15.1 (Summary) of Chap. 15 (Conclusion and Outlook)

For a more sophisticated insight we suggest the following reading order; it still skips technical details, but also provides an overview of *how to apply* our approach.

Introduction:
Chap. 1 (Introduction)
Chap. 2 (Informal Survey)
Formal Parts:
Informal survey and summary of chapters 3 (Modeling Object Contents and Digital Archives) to 7 (Incorporating Graph-Based Queries)

Case study:
Chap. 8 (Case Study)
Chap. 9 (Modeling Datatypes)
Chap. 10 (Implementing the Migration)
Introduction of Chap. 11 (Modeling Concepts)
Chap. 12 (Specifying and Checking Preservation Requirements)
Chap. 13 (Summary — Costs and Benefits)
Conclusions:
Chap. 15 (Conclusion and Outlook)

Readers who are interested in *mathematical details* should start with the "first reading". A general overview of our method is necessary to understand reasons for choosing exactly those formal techniques that we use. The formal chapters in part II (Formalization) depend on each other in ascending order of chapter numbers. Hence, readers should have read chapters 3 and 4 before reading Chap. 5. The chapters in Part III (Improving Usability) are mutually independent, but require knowledge of Part II. All "formal" chapters start with an informal or semi-formal survey. There, we match notions of the application domain to notions and techniques known in Mathematics and Computer Sciences. In the subsequent sections we then apply these techniques to the application domain. These parts are all formal. We use short examples to underline important aspects. For readability, we source out proofs to App. C. The proofs are linked in the

text. In some chapters we provide sophisticated semantics definitions (e.g., for concepts and preservation constraints). For these parts we provide continuative insights in App. B by fully formalizing parts of our running example; affected chapters include hints.

Chapter 2

Informal Survey

To start with, we give an overview of our approach and its application in a real-word setting. We first introduce a running example and then illustrate several facets of our approach and of the notion "preservation" on an informal level.

2.1 Running Example — Website Transformation

We use a structural website transformation as running example; the full case study will be presented in Chap. 8. Fig. 2.1 shows a simplified web archive and illustrate how the website transformation is integrated into an internal archival process. The web archive consists of two components. The *Archiver* includes a permanent storage for long-term preservation of the hosted websites. The servers being annotated with locks indicates that access to this storage environment is not permitted from "outside" the archive. The latter is provided by the *Browser*-component, which includes a web-storage with fast server access. The archive has two external communication interfaces – the customer interface (*CI*) and the user interface (*UI*). Customers can ingest a website to the archive, which corresponds to the event *INGEST*. The *Browser* component provides a service for full website browsing as well as quick search facilities. Users can request these services by issuing a *REQUEST*-event. Upon request, they will receive the corresponding data as a *RESPONSE*-event.

On ingest, the *Archiver* first extracts metadata and stores the uploaded website as well as the extracted metadata permanently (*extractMetadata*, *storePerm*). After

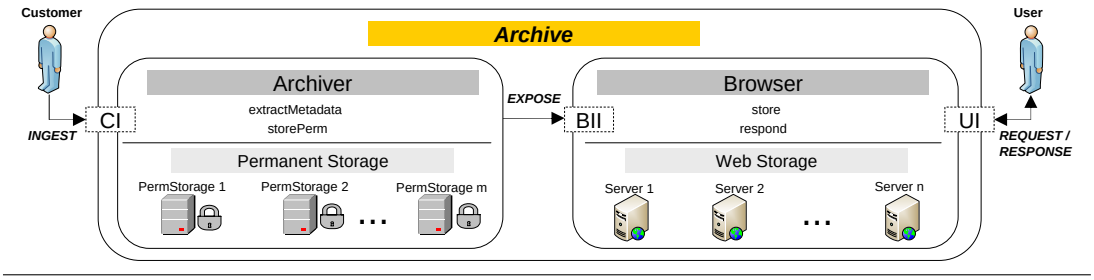


Figure 2.1: Example web archive

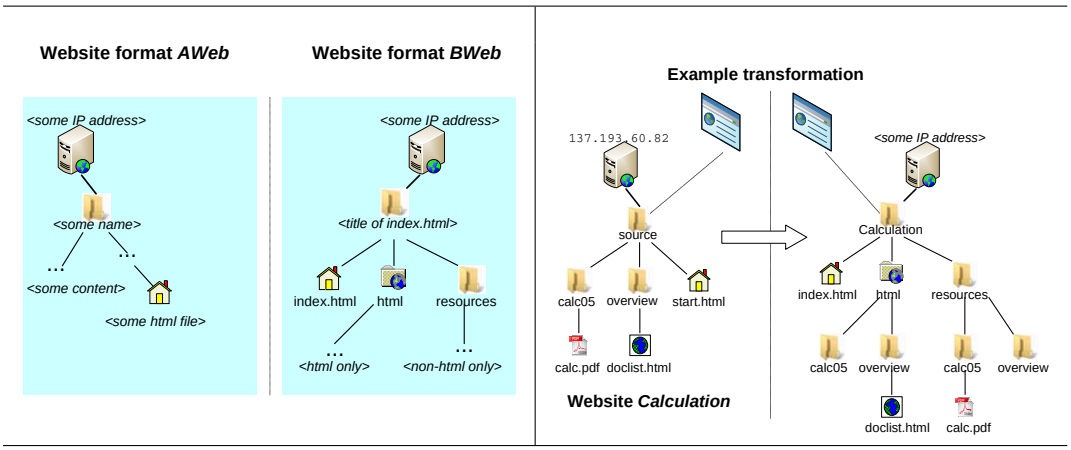


Figure 2.2: Website models and example transformation

that, the *Archiver* sends the internal event *EXPOSE* to the *Browser*'s internal interface (BII). The *Browser* then stores the website in the web storage (*store*). The *Archiver* and *Browser* use different website models for storage. The *EXPOSE* event, thus, includes a structural transformation of the underlying website.

Both the *Archiver*'s and *Browser*'s website model *AWeb* and *BWeb*, respectively, are depicted in the left-hand part of Fig. 2.2. The *Archiver* receives websites that have been harvested from some server or ingested by a customer and only requires them to contain a source directory and a welcome page ("home"-icon), which has to be located somewhere in the source directory. The *Browser* stores all websites on a web server and constrains directory content and structure of these websites in the following way:

(1) The source directory of the website exactly contains two sub-directories "html" and "resources", respectively, and a file named "index.html", which is the welcome page of the website.

(2) The names of the source directory and of the website are equal. If the website's welcome page has a `<title>` element with non-empty textual content, both names equal the content of this element.

(3) All html-files except "index.html" must reside in the directory "html", which is illustrated by the special directory icon.

(4) The directory "resources" contains all non-html-files.

In the right-hand part, Fig. 2.2 depicts an example instance of the standard website model *AWeb*, which resides on a server with IP 137.193.60.82 and is to be transformed to *BWeb*. The website "Calculation" consists of a directory structure containing two html-files and one pdf-file, where "start.html" is the welcome page. For this introductory survey we do not yet need the contents of the html-files. Yet we already mention that "start.html" contains the title of the page and a relative link to "doclist.html". Moreover, "doclist.html" includes a link to "calc.pdf". These "technical" properties will be studied in more detail in the technical part of this thesis.

The transformation from *AWeb* to *BWeb* is to adhere to the requirements shown in Fig. 2.3. The website's title mentioned in item (5) can be derived from its welcome

(1) The transformation result matches the *BWeb* format.
(2) Preserve file names as far as possible.
(3) Preserve directory names as far as possible.
(4) Preserve the website's name if possible.
(5) Preserve the title of the website.
(6) Preserve link consistency while transforming absolute to relative links.
(7) Preserve content and structure of the html-files as far as possible.
(8) Keep the bit-wise content of all non-html files unchanged.
(9) Preserve the directory and file structure of the source website in both the "html" and "resources" directory.

Figure 2.3: Preservation requirements for running example

page. Non-html content is separated from html content in the target model. However, item (9) assures that all html content in the "html" directory adheres to the directory structure of the source website. The same is true for all non-html content in "resources". In this way we avoid name conflicts. The website on the right-hand side of Fig. 2.2 is an appropriate transformation result of "Calculation" w.r.t. the requirements in Fig. 2.3.

However, up to now the preservation requirements are still expressed too informally for being verified automatically. While still remaining on an illustrative level, we will render some of them more precise in the following sections. In particular, we will (1) identify concepts for modeling these requirements, (2) illustrate different concept implementations in the source and target website models, (3) formulate preservation requirements more precisely, and (4) illustrate different variants of preservation and their relations to tracing object properties.

2.2 Concept Identification and Matching

According to our explanations in Sect. 1.2, the first step towards "trustworthy" automation is sufficient formalization. In order to keep this part as untechnical as possible, we postpone the formal parts to Chap. 3. We rather depict some of the properties of the requirement listing in the last section and illustrate how to apply our approach.

We start with preservation requirements (1) and (9) of Fig. 2.3 and show how to model the concepts related to them. Whenever a collection of files is ingested or harvested from the web, we have to identify it as a well-formed website. Our archive merely accepts websites that adhere to *AWeb*. For this purpose, we have to specify this format in more detail. The same is true for *BWeb*. First, we observe that both representations have a source directory that contains an html file. This html file is the entry point of the website. Additionally, both formats comprise a website object, which can be understood as being "abstract". In practice, it could be represented by some meta data containing the title of the website, for example (cf. [PRE05]).

These explanations exhibit an existing *semantic relationship* between these three entities. We capture this relationship by introducing a concept EntryPoint as shown

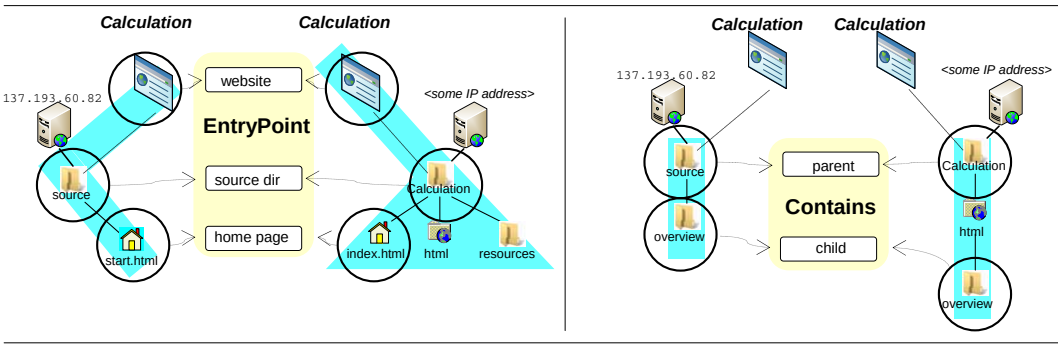


Figure 2.4: Example concepts and contexts

in the left-hand part of Fig. 2.4. The interface contains three *role names*: **website**, **source dir**, and **home page**. All implementations of EntryPoint have to assign these roles to concrete objects. These objects are called *interface objects* in the following. On the left-hand and right-hand side of the EntryPoint concept we show the relevant parts of the source and target website. The arrows show role assignment. The directory "source" of website "Calculation", e.g., takes the role **source dir**. So does the folder "Calculation" of the target website. Since all roles are assignable, we can *match* the concept EntryPoint to both the source and target website. However, this concept is implemented differently for *AWeb* and *BWeb*. We have indicated this by shaded areas. In particular, the target website must contain an "html" and a "resources" directory so as to conform to *BWeb*; this is not necessary for the source website. We say that the websites in Fig. 2.4 *match* EntryPoint in different *contexts*. This directly corresponds to our intention expressed in Sect. 1.2, where we stated that concepts *group* different implementations of one and the same aspect in different contexts.

In its right-hand part, Fig. 2.4 depicts another concept. It models directory containment and contains the roles **parent** and **child**. This concept is necessary for requirement (9) (preservation of directory structures). Again, the shaded areas indicate different implementations. In particular, the source directory may not contain any sub-directories other than "html" and "resources" in the *BWeb* format. Hence, all sub-directories must reside in one of these directories. Fig. 2.4 depicts containment in the "html" directory as an example.

Preservation requirements (2), (3), and (4) relate to an *attribute* of files, directories, and websites, respectively — they require the preservation of their *names*. This is modeled by a concept Name and the *value* of this concept must be identical for the source and target objects. As opposed to the semantic relationships in Fig. 2.4, different implementations can yield values other that "does hold" and "does not hold". In our approach we cover this difference by *non-functional* and *functional* concepts. EntryPoint is non-functional, Name is functional.

2.3 Preservation — Variants and Specification

Now that we have identified relevant concepts for the preservation requirements, we have to formulate the preservation requirements themselves in a machine-processable

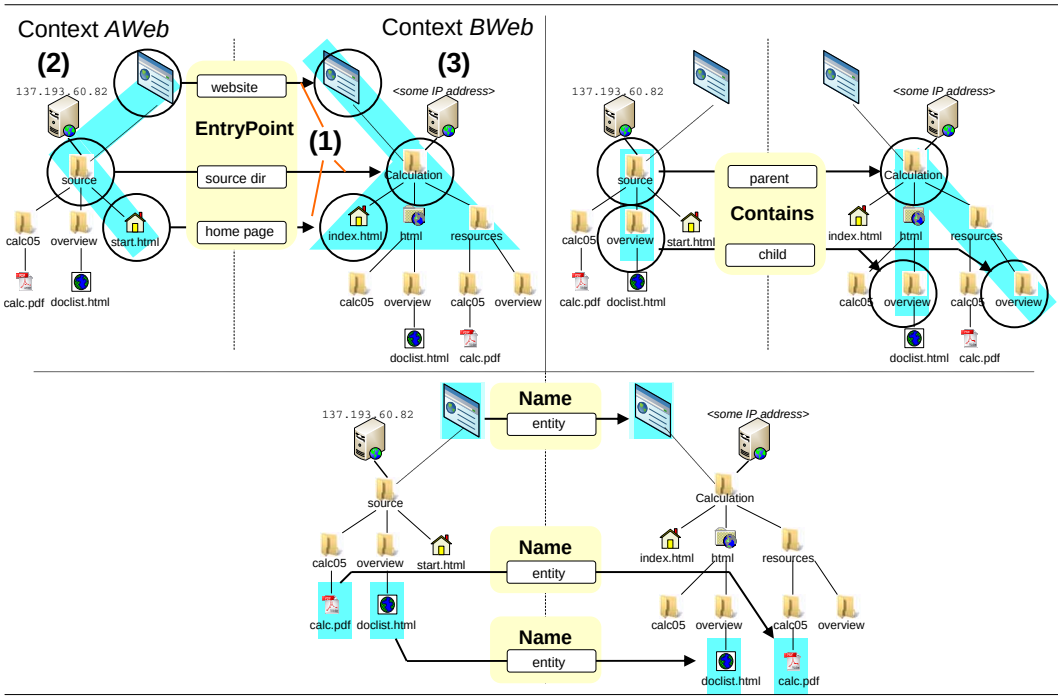


Figure 2.5: Different variants of preservation.

manner. Yet before we clarify the variants of preservation that are supported by our approach. During the following explanations one important aspect has to be kept in mind: According to the most relevant literature [Con02, PRE05], objects — once they are stored in an archive — are considered to be *immutable*. Whenever a change is to be made to an object, a new version of this object has to be generated (i.e., the object has to be *transformed*). This newly created object receives a new ID number, which is unique system-wide.

As indicated at the end of the last section, we support two variants of preservation. We call them *non-functional concept preservation* and *functional concept preservation*. Non-functional concepts can be used to preserve semantic relationships. Functional concepts embody object content. Hence, non-functional concept preservation corresponds to preserving object content.

In its upper left-hand part, Fig. 2.5 illustrates the preservation of the non-functional concept EntryPoint. We have highlighted three important aspects related to the source and target website using marks (1), (2), and (3).

- We trace the transformation of those objects that are assigned to role names of the concept that is to be preserved (mark (1)). Transformations of other elements are irrelevant for the preservation of this concept. This directly corresponds to our view on a concept's interface — only objects taking roles in a concept definition are considered to be relevant for this concept. Keeping in mind that all objects are immutable, only transformations can cause changes by creating new object versions. This is why we strictly relate the notion of preservation to transformations.
- Whenever a transformation of the interface objects is recognized, we check whether

concept satisfaction in the source context *AWeb* (shaded area marked with (2)) correlates with concept satisfaction in the target context *BWeb* (shaded area marked with (3)). Only if both checks result in the same truth-value, this concept is preserved.

The bottom part of Fig. 2.5 illustrates the preservation of the functional concept Name. Websites, directories, and files have a *name*-property (or attribute) the value of which has to be preserved by the transformation. This concept is preserved if the source and target object have equal names. Again, object immutability requires transformation if a change is desired. Notice the strong correlation between functional and non-functional concept preservation. In order to satisfy the concept EntryPoint in the target concept, the **home page** file is always named *index.html*. This prevents the preservation of the name property in our example since the **home page** file of the source website is named *start.html*. Therefore the corresponding preservation requirement has been expressed by "Preserve file names *where possible*".

Up to now, we have considered "linear" object histories only. The upper right-hand part of Fig. 2.5, however, shows an example where we produce different versions of certain directories. In particular, preservation requirement (9) demands the preservation of the full directory and file structure of the source website in both the "html" and "resources" directory of the target website. Concerning the directories "calc05" and "overview", this yields two new versions for each. According to this preservation requirement we also have to preserve containment of the file "doclist.html" in "overview" and "calc.pdf" in "calc05", respectively. One may have realized that this preservation requirement can be satisfied for one new version of the corresponding directories, only. The reason is that "html" may contain hypertext content only. Analogously, "resources" may contain non-hypertext only. These different preservation requirements are to be covered by our approach.

With these remarks we have already made a step towards expressing preservation requirements in a more precise way. In the technical part of this thesis we will introduce formal *preservation constraints* the verification of which can be done automatically. In order to get an understanding of the way we use these formal constraints, we start with a first progress from the informal requirements of Fig. 2.3 towards semi-formal requirements, which are expressed more precisely. Recall that preservation requirement (1) is simply expressed by "The transformation result matches the BWeb format." in Fig. 2.3. We have modeled this property using EntryPoint. According to our explanations so far, we directly relate our notion of concept preservation to the concept itself, to the transformation of the concept's interface objects, and to a source and target context. Therefore, we formulate this preservation requirement more precisely as follows:

> When transforming an object, which is assigned to one of the roles **website**, **source dir**, or **home page** of the concept EntryPoint, **all new versions** of all interface objects of EntryPoint must **satisfy the concept** EntryPoint **in** the **target context** *BWeb*, **if and only if** the **source objects satisfied** EntryPoint **in** the **source context** *AWeb*.

Two parts of this sentence require for explicit attention:

(1) We want *all* new versions of the source objects to satisfy this requirement. The reason for this explicit distinction has been explained before (using the Contains concept of Fig. 2.5).

(2) We postulate the strong correspondence "if and only if" between concept satisfaction in the source and target context, respectively. This automatically includes that the transformation results must also *not* satisfy the concept EntryPoint if the source objects did not. This corresponds to a rigorous understanding of preservation: We want to preserve the *status* of concept satisfaction w.r.t. the concept EntryPoint. In our view, this includes both cases — satisfaction and failure.

The above semi-formal formulation of preservation requirement (1) is sufficiently precise in the sense that it expresses the following five intentions:

(1) Concept preservation for all histories —
"**... for all new versions ...** "

(2) Relation to transformation of interface objects —
"**When transforming ...** "

(3) Relation to concept and its interface objects —
"**... source objects satisfied** EntryPoint**...**"

(4) Definition of permitted source contexts —
"**satisfied** EntryPoint **in the source context** *AWeb*"

(5) Definition of permitted target contexts —
"**satisfy ...** EntryPoint **in the target context** *BWeb*"

In particular, we permit different implementations of one and the same aspect in different contexts, while nothing is said about *how* the contexts *AWeb* and *BWeb* really implement the interface of EntryPoint. All we know is that they comprise three interface objects that have semantic relationships. Hence, changing implementations do not necessarily induce a change to requirements specifications. This interface-based approach is a well-established technique in the context of programming. Additionally, our preservation *language* supports readability of the preservation requirements even for those that have no or limited background in Mathematics or Computer Science. What is still missing, is a precise, machine-readable syntax as well as a formal semantics for expressing and evaluating preservation requirements. We shall see that the syntax is a straightforward technical realization of the formulation above. According to our explanations above, the semantics for constraint evaluation has to take into account operations (like a transformation) and concept evaluation for different implementations in possibly different system states. The technical details will be explained in Part II. In the following section we briefly sketch how our framework works in terms of automated verification of preservation requirements.

2.4 Verifying Preservation Requirements

In the following we demonstrate the state-based functionality of our framework. We transform parts of the source website of Fig. 2.2 step by step and explain what is happening "behind the curtain". In our explanations, we always refer to the following two preservation requirements:

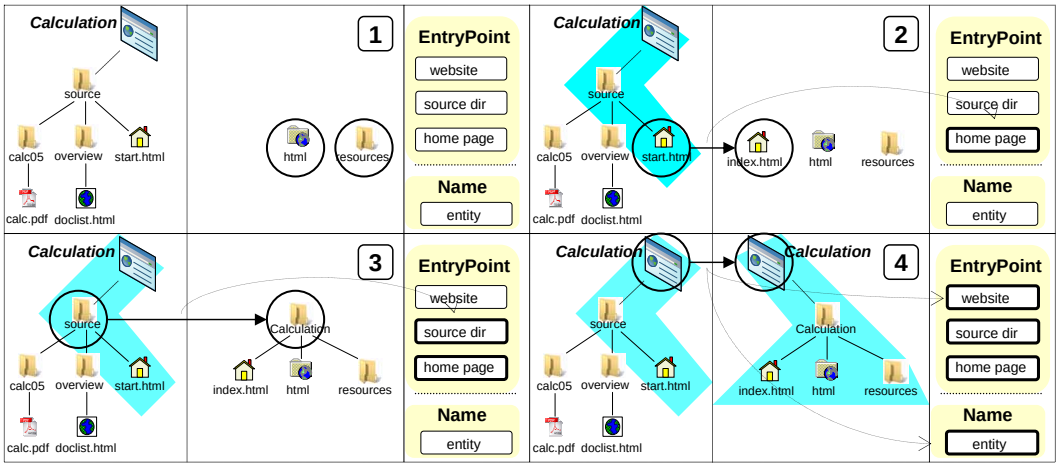


Figure 2.6: State-based transformation and verification.

(1) *Preservation of concept* EntryPoint*:* When transforming an object, which is assigned to one of the roles **website**, **source dir**, or **home page** of the concept EntryPoint, all new versions of all interface objects of EntryPoint must satisfy the concept EntryPoint in the target context *BWeb*, if and only if the source objects satisfied EntryPoint in the source context *AWeb*.

(2) *Preservation of concept* Name*:* When transforming a website object, which is assigned to the role **entity** of the concept Name, the value of the concept Name in context *WN* for all new versions of the interface object must equal the value of the concept Name in context *WN* for the source object.

In Fig. 2.6 we show four transformation steps. In step one (upper left-hand part), we create two directories "html" and resources, respectively. Next to this box we have indicated the preservation requirements by the interfaces of EntryPoint and Name. These two operations do not affect any preservation requirement so far as our notion of preservation is related to transformations only. Nevertheless, the migration process has to communicate these operations to our framework as they are relevant when checking the target website for *BWeb* conformity. So far, our system has recorded the *migration sequence* ⟨`create(html),create(resources)`⟩. In the next step, the welcome page of the source website is transformed to "index.html". Again, this operation is communicated to the system, which yields the overall migration sequence

⟨`create(html),create(resources),transform(start.html -> index.html)`⟩.

The file "start.html" is the welcome page of the source website and, hence, meets the prerequisite "When transforming an object, which is assigned to one of the roles **website**, **source dir**, or **home page** of the concept EntryPoint.." for the application of preservation requirement (1) from above. This activates concept matching, which is shown by the shaded area in the upper right-hand part of Fig. 2.6. Our system is now aware of this transformation and marks the corresponding role "activated" for the concept EntryPoint, the source object, and the target object. The same is done in step three with the directory "source" and role **source dir**. Hence, in state three, two out of three roles are

activated of EntryPoint's interface. In step four, the website object is transformed. This activates the role **website**. Additionally, this transformation has to preserve the name of the website object. The overall migration sequence becomes

```
⟨create(html),create(resources),transform(start.html -> index.html),
transform(source -> Calculation), transform(Calculation -> Calculation)⟩.
```

The transformation is finished in state four. At this point, our system has to be notified in order to activate the verification of the underlying preservation requirements. In our example, the target website satisfies the concept EntryPoint in the context *BWeb*, which means that the first preservation requirement is satisfied. Also, the website's name is preserved, which satisfies requirement (2) from above. If one of the requirements was not satisfied, our system would precisely report (1) which preservation requirement has not been met, and (2) which objects are to blame for this violation. This allows for adequate reactions.

2.5 Summary

We have surveyed our approach in an informal manner. A website transformation application has been introduced as running example. It exhibits all aspects that are necessary to understand our approach. Although this example may seem to be simplistic, we shall see that the formal treatment of some preservation requirements (like link consistency) is not simple at all.

To start with, we have introduced the corresponding preservation requirements on an informal level. Afterwards, we have demonstrated three steps towards automating the verification process of these requirements:

(1) Define relevant properties (i.e., concepts) related to the preservation requirements.

(2) Express the preservation requirements in a sufficiently formal way.

(3) Verify these preservation requirements in a state-based environment.

In that, we have identified two variants of preservation, namely functional and non-functional concept preservation. The latter handles the preservation of semantic object *relationships* that may be implemented differently for the source and target objects. In contrast, functional concept preservation handles object *content*. Preservation in this respect means identical preservation of the *value* of the functional concept.

We have progressed from informal to semi-formal formulations of preservation requirements. Semi-formal formulations precisely relate source and target objects to concepts that are to preserved in (possibly different) source and target contexts. This coincides with the basic intention behind our notions of preservation. Hence, there is only a small step to translating semi formal requirements to machine-processable preservation constraints.

Finally, we have used excerpts of the example transformation to demonstrate the operation of our framework. In order to automate the verification process, our system must be notified on state changes like object creation or object transformation. It then

records these *migration sequences* and particularly uses transformations for tracing object histories. That way we can relate source objects to their new versions and, thus, check preservation requirements in an automated way.

Part II

Formalizing Migration and Preservation

Chapter 3

Modeling Objects and Digital Archives

Here we introduce a basic dynamic environment for Digital Archives (DA). It (1) covers datatypes for digital objects as well as functions and semantic relationships defined on them and (2) captures the effects of migration processes by well-defined basic state changes. With these preliminaries we can introduce formal concepts (Chap. 4) and our preservation language (Chap. 5); formal preservation requirements can be evaluated in the basic DA.

The outline is as follows:

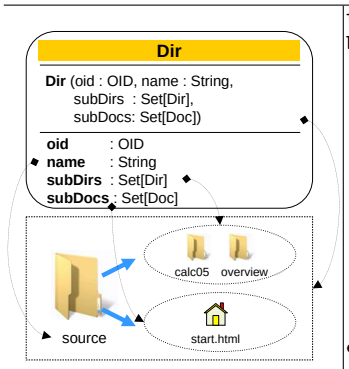
- An informal survey is provided in Sect. 3.1.
- Object content and object relationships are modeled in Sect. 3.2. We motivate why we use Abstract Data Types (ADTs) and define the necessary formal structures.
- In Sect. 3.3 we integrate these (static) structures into a dynamic environment. We define a basic formal DA. It includes predefined administrative components that suffice to characterize the content of digital objects formally. Sequences of formal state transitions (object creation, object transformation, object deletion) cover migration processes. These state transitions will be characterized by pre- and post conditions; this facilitates to prove some important system invariants. A reference implementation is provided in Chap. 6 by means of Abstract State Machines (ASMs).
- We close with a short summary in Sect. 3.4.

All formal parts will be accompanied by non-formal explanations related to our running example.

3.1 Informal Overview

According to our explanations in Chap. 1, we have to obey some constraints when modeling DAs:

- DAs usually comprise administrative components that serve the preservation of their hosted digital objects (cf. Fig. 1.3, page 6). Hence, predefined functionality




```
type Dir<{DObj} =
begin
  constr [ Dir : OID × String × Set[Dir] × Set[Doc] → @
  ops    [ oid      : @ → OID
           name     : @ → String
           subDirs  : @ → Set[Dir]
           subDocs  : @ → Set[Doc]
  axioms [ ∀oid : OID, name : String, subDirs : Set[Dir],
             subDocs : Set[Doc] •
           oid(Dir(oid, name, subDirs, subDocs)) = id∧
           name(Dir(oid, name, subDirs, subDocs)) = n∧
           subDirs(Dir(oid, name, subDirs, subDocs)) = subDirs∧
           subDocs(Dir(oid, name, subDirs, subDocs)) = subDocs
end Dir;
```

Figure 3.1: Object type `Dir` as UML-like visualization and ADT specification

like validity or integrity checks must be supported.

- Objects must be uniquely identifiable and their content must be accessible.
- DAs evolve due to running migration processes that create, delete, or transform digital objects.
- Objects are immutable; content changes result in new objects with a new ID. We, thus, have to characterize object content.

We specify object types, object contents, and archival functionality as *Abstract Datatypes* (ADT,[CoF04, EGL89, Wir95, ABK+02]). In Fig. 3.1 we show, a UML-like visualization (left-hand part) and an algebraic ADT specification (right-hand part) of the *object type* `Dir` (directories). This pseudo specification language will be used in the sequel and is very much related to the CASL specification language [BM04]. It, however, emphasizes our focus on datatypes. The `constr`-part specifies constructors, `ops` specifies operations, and the `axioms` part fixes all axioms. The symbol @ means "self", the preamble `type Dir<{DObj}` states that `Dir` is a subtype of type `DObj`; `DObj` is the super-type of all digital objects.

Objects are constructed using distinguished *constructors* which we basically handle like regular functions. However, they will be used for characterizing object contents. According to Fig. 3.1, `Dir` constructs directories. The axioms on the right-hand side show that the parameters *oid* (object ID), *name* (name), *subDirs* (sub-directories), and *subDocs* (sub-documents) fix the value of the corresponding *attributes* [1] (`oid`, `name`, `subDirs`, and `subDocs`). These attributes describe named, tree-like directory structures (see bottom left-hand part of Fig. 3.1); generally speaking, attributes reflect *accessible* object content. Type `Set` shows that we support generic types; this supports re-use.

ADTs are a key to our approach. They are defined inductively and allow for equality-based reasoning. Inductive structures (like trees) frequently occur in our setting. Also, we will introduce an equality-based notion of preservation in Chap. 5.

Notice the difference between object IDs in our setting and object *references* (e.g., in object-oriented systems). We include IDs as part of object content; they are *persistent*. As objects are immutable, object IDs cannot be changed as well. This helps maintaining

[1] In the context of ADT specifications, attributes are rather called *selectors*. In the DA community, however, the term *attribute* is more common; we adapt the latter.

object integrity. In contrast, references are mostly not durable over the long term but may *include* object ID (like web-URLs that include document identifiers (DOI)).

Our basic DA includes the following administrative components:

The datatype `OID`*:*
It models (an infinite set of) IDs that can be attached to objects on creation; given an ID *id*, the next valid ID can be computed by `nextID`(*id*).

The datatype `DObj`*:*
`DObj` is the supertype of all digital objects; types being no subtype of `DObj` are called *basic types*.

The function `existDObj`:
It stores the set of *existing* objects. On object creation `existDObj` is extended; on object deletion it is reduced.

The function `usedOIDs`
It stores all object IDs that have ever been used. These IDs are never used again.

Basic types and functions belong to the *static* parts of an archive. In particular, the semantics of static functions (like `nextID`) never changes. In contrast, the semantics of dynamic functions (`existDObj`, `usedOIDs`) can evolve.

Example 3.1.1 (Effect of basic state changes)

Tab. 3.1 depicts a sample archive evolution; we create the "source" directory of our example website (Fig. 2.2, page 17) and delete it again. The resulting state changes are visualized in the upper part. The table rows contain the contents of `usedOIDs` (column two) and `existDObj` (column three) in the respective system state (column one). `HTMLDoc` constructs html files; the terms are abbreviated for the sake of simplicity.

In concrete syntax, the corresponding basic state changes are given as follows:

(1)	$x_0 := \mathsf{cre}(\texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, ...))$	(State **0**)
(2)	$x_1 := \mathsf{cre}(\texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\}))$	(State **1**)
(3)	$x_2 := \mathsf{cre}(\texttt{Dir}(\texttt{nextID}^2(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{\}))$	(State **2**)
(4)	$x_3 := \mathsf{cre}(\texttt{Dir}(\texttt{nextID}^3(\texttt{initID}), \text{“}\texttt{source}\text{“}, \{x_1, x_2\}, \{x_0\}))$	(State **3**)
(5)	$\mathsf{del}(x_2)$	(State **4**)
(6)	$\mathsf{del}(x_3)$	(State **4**)

We have assigned newly created objects to variables x_0 to x_3 in order to abbreviate notation.

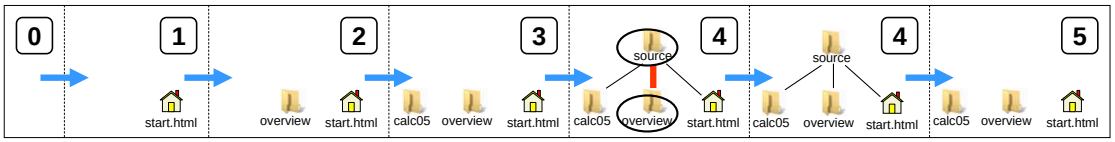
□

In state zero `usedOIDs` and `existDObj` are empty; no objects exist. After that, "start.html", "calc05", and "overview" are created successively. The values of `existDObj` and `usedOIDs` adapt adequately. Notice that we require all object constructors to have an ID parameter in the first position; all objects contain a persistent identifier.

In state three we create the directory "source"; it contains x_1 ("calc05") and x_2 ("overview") as sub-directories (attribute `subDirs`) and x_0 ("start.html") as sub-document (attribute `subDocs`).

In the next step we try to delete "overview"; this, however, is blocked. As "overview" is a sub-directory of "source", it belongs to the *object-valued content* of "source"; deleting "overview" would change the content of "source". As objects are immutable, this operation has no effect. This is assured fully automatically by our system.

Table 3.1: *Example archive evolution*



State	Value `usedOIDs`	Value `existDObj`
0	$\{\}$	$\{\}$
1	$\{\texttt{initID}\}$	$\{\texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, \ldots)\}$
2	$\{\texttt{initID}, \texttt{nextID}(\texttt{initID})\}$	$\left\{\begin{array}{l}\texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, \ldots)\\ \texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\})\end{array}\right\}$
3	$\left\{\begin{array}{l}\texttt{initID}, \texttt{nextID}(\texttt{initID}),\\ \texttt{nextID}^2(\texttt{initID})\end{array}\right\}$	$\left\{\begin{array}{l}\texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, \ldots)\\ \texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\})\\ \texttt{Dir}(\texttt{nextID}^2(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{\})\end{array}\right\}$
4	$\left\{\begin{array}{l}\texttt{initID}, \texttt{nextID}(\texttt{initID}),\\ \texttt{nextID}^2(\texttt{initID}), \texttt{nextID}^3(\texttt{initID})\end{array}\right\}$	$\left\{\begin{array}{l}\texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, \ldots)\\ \texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\})\\ \texttt{Dir}(\texttt{nextID}^2(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{\})\\ \texttt{Dir}(\texttt{nextID}^3(\texttt{initID}), \text{“}\texttt{source}\text{“}, \{x_1, x_2\}, \{x_0\})\end{array}\right\}$
4	$\left\{\begin{array}{l}\texttt{initID}, \texttt{nextID}(\texttt{initID}),\\ \texttt{nextID}^2(\texttt{initID}), \texttt{nextID}^3(\texttt{initID})\end{array}\right\}$	$\left\{\begin{array}{l}\texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, \ldots)\\ \texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\})\\ \texttt{Dir}(\texttt{nextID}^2(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{\})\\ \texttt{Dir}(\texttt{nextID}^3(\texttt{initID}), \text{“}\texttt{source}\text{“}, \{x_1, x_2\}, \{x_0\})\end{array}\right\}$
5	$\left\{\begin{array}{l}\texttt{initID}, \texttt{nextID}(\texttt{initID}),\\ \texttt{nextID}^2(\texttt{initID}), \texttt{nextID}^3(\texttt{initID})\end{array}\right\}$	$\left\{\begin{array}{l}\texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, \ldots)\\ \texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\})\\ \texttt{Dir}(\texttt{nextID}^2(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{\})\end{array}\right\}$

In the last step, we delete “source”. There, `usedOIDs` stays unchanged. However, the just-described content-relationship is released; x_0, x_1, or x_2 could now be deleted as well.

A state change $\texttt{trans}(t_{src} \mapsto t_{trg})$ models an object transformation. Since our basic DA does not store object histories explicitly, this state change basically has the same effect on `existDObj` as creating t_{trg} using $\texttt{cre}(t_{trg})$. However, transformations are used to *trace* object histories in the course of migration processes. They will play an important role for our notion of preservation later on.

To sum up, our basic archive satisfies the following properties (cf. Tab. 3.1):

- All objects in `existDObj` have different IDs.
- Whenever an object is in `existDObj`, all its object-valued content is in `existDObj`.
- Object creation/transformation extends `existDObj`, if all object-valued content of the candidate exists; otherwise the operation has no effect.
- Object deletion reduces `existDObj`, if the candidate is not in the content of another existing object; otherwise the delete operation does nothing.
- Object IDs are used once only.

In the following sections we will provide the necessary formalisms.

3.2 Modeling Object Contents and Relationships

Our approach is to capture the preservation of properties related to *digital objects*. We, thus, first have to describe object content and object relationships. Before we introduce the formalisms, we want to mention the key observations that drove our decision:

- In our application domain, digital objects are usually *classified*, although this classification may vary between different layers of abstraction. Document formats are one particular example. Therefore, a typing mechanism for objects is necessary.
- We have to model functionality defined for objects. In the last section we mentioned validity checks. This functionality is defined w.r.t. an object's content and usually has certain formal properties. An html document starting with another than an `<html>`-tag is no valid html document. A validity check must not return `True` in that case. The methodology has to offer facilities to express these properties.
- Object relationships are important. As an example, (recursive) directory containment occurred in the example survey of Sect. 2.1. All these properties do not depend on the evolving system but can be specified statically.

These aspects must be covered with sufficient formal clarity in order to meet our claim for high trustworthiness. Given these requirements, the literature offers two standard methods that are worth being considered in more detail. First, object-oriented specification techniques are strongly supported by the Unified Modeling Language [HKKR05]. Typing is implemented by classes. The content of digital objects is captured by attributes. Invariants or specific restrictions can be specified using the Object Constraint Language (OCL, [Ric02]). Object relationships are modeled by associations and multiplicities, but can also be refined using set-valued attributes etc. The sophisticated tool-support including graphical visualization clearly eases understanding of the described models. Yet there is no substantial tool-support available for automated reasoning w.r.t. UML specifications. This has two reasons: First, the UML-semantics itself is not specified formally. Second, UML aims at *stateful* objects, i.e., such changing their attribute values over lifetime. The semantics of an object is very complex as it covers a full state space. Function calls work by reference (object reference), the value then is dynamically determined in the system state the function call is executed in. Therefore, functional properties are usually specified by pre- and post conditions. In our setting, however, we focus at immutable (i.e., *stateless*) objects.

As opposed to the UML-approach, algebraic specifications of Abstract Data Types [EGL89, Wir95] do not inherently cover stateful objects. They are rather used to specify a type domain by fixing all properties that instances of this domain have to satisfy. Datatypes are defined inductively by constructors, which facilitates induction proofs. Since function calls work by value, properties of functions and relationships (predicates) can be specified using equality, which allows for induction proofs based on equality and term-rewriting. The ADT-approach is, thus, optimized for stateless reasoning. It has been pushed forward over the last years resulting in growing tool-support. The CASL specification language ([BM04, CoF04]) is a prominent example. It is integrated into the powerful theorem prover *Isabelle* ([Isa02]) and, hence, offers support for automated reasoning. With the extension *HasCASL*, implementations for the functional programming language Haskell ([Tho99]) can be derived automatically. Algebraic ADT specifications base on formal model theory. All specifications have a clear semantics, but strong expert skills are necessary in order to pervasively understand this specification method. Also, the notion of *object-valued content* has to be characterized separately; the ADT-approach does not cover attributes innately. Yet the following criteria drove our decision to use algebraic ADTs:

(1) The approach offers a sound semantics. This prevents ambiguity and facilitates to apply formal proof techniques in the domain of digital preservation.

(2) Inductive type definitions frequently occur in the context of digital preservation. As an example, most document formats are tree- or graph based. Also, directory structures are trees. Available tools can, thus, be used for automated, inductive reasoning in our application domain.

(3) Digital objects are immutable, thus, stateless.

These observations are particularly confirmed by our case study (Part IV). In order to ease the understanding of datatype specifications, we will often represent them using the UML-like graphical visualization shown in the upper left-hand part of Fig. 3.1.

Incorporating formal datatype specifications requires some effort. In the following section we introduce the necessary formal structures and define datatype specifications formally.

Formal Structures for Datatype Specifications

We start by introducing some notational conventions. Throughout this thesis, we will often deal with calculi that can be used to derive certain properties. For this purpose, we introduce the rule notation

$$\frac{\begin{array}{cc}\phi_1 & \phi'_1\\ \ldots & \ldots\\ \phi_m & \phi'_n\end{array}}{\psi}$$

It reads as follows: The assumption ψ can be inferred if the judgments ϕ_1 through ϕ_m and ϕ'_1 through ϕ'_n hold. This introduces a derivability relation (which is often denoted by $\vdash$ in the literature). Notice that we will always refer to the *least* fitting relation that satisfies the specified rules. As an example, we will introduce a calculus for deriving the semantic value of a syntactic term. This will be denoted by $(\mathcal{A}, \eta, t) \overset{t}{\rightsquigarrow} v$ where $\mathcal{A}$ is an algebra, η a variable assignment, t a term of a given signature, and v the resulting semantic value of t. The $\rightsquigarrow$ operator will occur in different variants and is always used to compute semantic values of syntactic elements. We, e.g., use $\overset{op}{\rightsquigarrow}$ for the derivation of the semantic effect of basic operations or $\overset{mig}{\rightsquigarrow}$ for the effect of migration algorithms (the latter occurs in Chap. 6).

We start by defining the syntax for ADT specifications by means of signatures.

Definition 3.2.1 (Signature Σ) A signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$, $\mathcal{C} \subseteq \mathcal{F}$, consists of

- a set of *type symbols* $\mathcal{T}$, where $\mathcal{T}^*$ denotes the set of all ordered type tuples $\overline{\tau_i} := (\tau_1, \tau_2, ...\tau_n)$ over $\mathcal{T}$, $1 \leq i \leq n \in \mathbb{N}$,
- a pre-ordering as *subtype relation* $<$, which we extend to a relation over $\mathcal{T}^* \times \mathcal{T}^*$ by $\overline{\tau_{1,i}} < \overline{\tau_{2,i}}$ $(i \in \{1, ..., n\})$, iff[2] $\tau_{1,1} < \tau_{2,1}, ..., \tau_{1,n} < \tau_{2,n}$,

[2] iff = if and only if

- a set $\mathcal{P}$, which is partitioned by sets $\mathcal{P}_{\overline{\tau_i}}$, $i \in \{1, ..., n\}$, containing finitely typed *qualified predicate symbols* $p : \tau_1 \times ... \times \tau_n$, $\tau_1, ..., \tau_n \in \mathcal{T}$ including a set of distinguished, overloaded instance predicates $\{\texttt{inst}_\tau : \tau' \mid \tau, \tau' \in \mathcal{T}\}$,
- a set $\mathcal{C}$, which is partitioned by non-empty sets $\mathcal{C}_\tau$ for each $\tau \in \mathcal{T}$ containing the *type constructors* $c : \overline{\tau_i} \to \tau \in \mathcal{F}_{\overline{\tau_i} \to \tau}$ for τ, and
- a set $\mathcal{F}$, which is partitioned by sets $\mathcal{F}_{\overline{\tau_i} \to \tau}$, $i \in \{1, ..., n\}$ containing finitely typed *qualified function symbols* $f : \tau_1 \times ... \times \tau_n \to \tau$, τ_1,...,τ_n,$\tau \in \mathcal{T}$.

Given a predicate symbol $p : \overline{\tau_i}$, p is the *predicate name* and $\overline{\tau_i}$ is the *predicate signature*. Analogously, f is the *function name* and $\overline{\tau_i} \to \tau$ the *function signature* for a function symbol $f : \overline{\tau_i} \to \tau$. We require predicate names, and function names to be pairwise disjoint sets not including $=$, i.e., $N_\mathcal{P} \cap N_\mathcal{F} = \emptyset$, and $= \notin N_\mathcal{P} \cup N_\mathcal{F}$, where $N_\mathcal{P} = \{p \mid n \in \mathbb{N}, p : \tau_1 \times ... \times \tau_n \in \mathcal{P}\}$, $N_\mathcal{F} = \{f \mid n \in \mathbb{N}, f : \tau_1 \times ... \times \tau_n \to \tau \in \mathcal{F}\}$. The set $|\Sigma|$ of symbols of Σ is defined by $|\Sigma| := \mathcal{T} \cup N_\mathcal{P} \cup N_\mathcal{F} \cup \{=\}$. □

Notice that we support an explicit `inst` predicate that is supposed to hold whenever the value of a term is in the respective type domain. In the following we will often use the short-hand notation $p_{\overline{\tau_i}}$ and $f_{\overline{\tau_i} \to \tau}$ for predicate symbols $p : \overline{\tau_i}$ and function symbols $f : \overline{\tau_i} \to \tau$, respectively.

Notice that we permit multiple signatures $\overline{\tau_i} \to \tau$ for the same function/predicate name f/p. Yet we still forbid mixing predicate names and function names. This will be important later on when we introduce a functional language for programming migration algorithms. In this language predicates can be used as regular Boolean-valued functions, which might lead to name conflicts.

In the following we will introduce algebras that assign interpretations to each function/predicate symbol. In conjunction with multiple inheritance, this symbol overloading may lead to ambiguities [CoF04]. Drastic solutions comprise excluding multiple inheritance or demanding globally unique function names. We argue that both are not practical in our setting. Globally distinct function names lead to "unreadable" specifications because equal aspects are expressed differently. Also, multiple inheritance occurs frequently in our application domain. In the case study (Chap. 9) we present a design pattern that bases on multiple inheritance. It can be used to trace structures *partially* and, thus, keep the number of objects in the system to a minimum. Also, there are usually different *views* on objects. This can be implemented by multiple inheritance in a comfortable way.

We adopt the solution of [CoF04] and identify an overloading relation and constrain algebras appropriately w.r.t. this relation.

Definition 3.2.2 (Overloading relation) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$. Two predicate symbols $p_{\overline{\tau_{1,i}}} \in \mathcal{P}$ and $p_{\overline{\tau_{2,j}}} \in \mathcal{P}$ are in the overloading relation (denoted by $p_{\overline{\tau_{1,i}}} \cong_\mathcal{P} p_{\overline{\tau_{2,j}}}$, iff there is a $\overline{\tau_k} \in \mathcal{T}^*$ such that $\overline{\tau_k} < \overline{\tau_{1,i}}, \overline{\tau_{2,j}}$.
Two function symbols $f_{\overline{\tau_{1,i}} \to \tau_1} \in \mathcal{F}$ and $f_{\overline{\tau_{2,j}} \to \tau_2} \in \mathcal{F}$ are in the overloading relation (denoted by $f_{\overline{\tau_{1,i}} \to \tau_1} \cong_\mathcal{F} f_{\overline{\tau_{2,j}} \to \tau_2}$), iff there are $\overline{\tau_k} \in \mathcal{T}^*$ and $\tau \in \mathcal{T}$ such that $\overline{\tau_k} < \overline{\tau_{1,i}}, \overline{\tau_{2,j}}$ and $\tau_1, \tau_2 < \tau$. □

The so-declared $\cong$ relation can be used to characterize unambiguous interpretations of overloaded function symbols in algebras $\mathcal{A}$.

Definition 3.2.3 (Σ-algebra) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$. A Σ-*algebra* $\mathcal{A} := (\mathcal{T}^{\mathcal{A}}, \mathcal{P}^{\mathcal{A}}, \mathcal{C}^{\mathcal{A}}, \mathcal{F}^{\mathcal{A}})$ consists of

- a non-empty *carrier set* $\tau^{\mathcal{A}}$ for each type $\tau \in \mathcal{T}$ such that

  (1) $v \in \tau^{\mathcal{A}} \cap \tau'^{\mathcal{A}}$ implies that there is $\tau'' < \tau, \tau'$ such that $v \in \tau''^{\mathcal{A}}$, and

  (2) $\tau < \tau'$ implies $\tau^{\mathcal{A}} \subseteq \tau'^{\mathcal{A}}$

  for all $\tau, \tau' \in \mathcal{T}$,
- a *predicate* $p^{\mathcal{A}}_{\tau_1 \times ... \times \tau_n} \subseteq \tau_1^{\mathcal{A}} \times ... \times \tau_n^{\mathcal{A}}$ for each qualified predicate symbol $p_{\tau_1 \times ... \times \tau_n}$ in $\mathcal{P}_{\overline{\tau_i}}$, where $\mathtt{inst}^{\mathcal{A}}_{\tau} = \tau^{\mathcal{A}}$ for all $\tau \in \mathcal{T}$,
- an element $c^{\mathcal{A}}$ in $\tau^{\mathcal{A}}$ for each constant constructor $c : \tau \in \mathcal{C}_{\tau}$, and
- a *partial function* $f^{\mathcal{A}}_{\tau_1 \times ... \times \tau_n \to \tau} : \tau_1^{\mathcal{A}} \times ... \times \tau_n^{\mathcal{A}} \to \tau^{\mathcal{A}}$ for each qualified function symbol $f_{\tau_1 \times ... \times \tau_n \to \tau}$ in $\mathcal{F}_{\overline{\tau_i} \to \tau}$. We define the *strict extension* $f^{\mathcal{A}^{\perp}} : \tau_1^{\mathcal{A}} \cup \{\perp\} \times ... \times \tau_n^{\mathcal{A}} \cup \{\perp\} \to \tau^{\mathcal{A}} \cup \{\perp\}$ of a partial function $f^{\mathcal{A}}$ by

$$f^{\mathcal{A}^{\perp}}(v_1, ..., v_n) = \begin{cases} \perp, & \text{any of the } v_i \text{ is } \perp \\ f^{\mathcal{A}}(v_1, ..., v_n), & \text{otherwise} \end{cases} .$$

  Given $\overline{\tau_k} \in \mathcal{T}^*$ $(1 \leq k \leq n)$, we also denote $\tau_1^{\mathcal{A}} \times ... \times \tau_n^{\mathcal{A}}$ by $\overline{\tau_k}^{\mathcal{A}}$ and require predicate and function interpretations to respect the overloading relation:

  - Whenever $p_{\overline{\tau_{1,i}}} \cong_{\mathcal{P}} p_{\overline{\tau_{2,j}}}$, and $\overline{\tau_k} < \overline{\tau_{1,i}}, \overline{\tau_{2,j}}$ then for all $\overline{v_k} \in \overline{\tau_k}^{\mathcal{A}}$ it is true that $\overline{v_k} \in p^{\mathcal{A}}_{\overline{\tau_{1,i}}} \Leftrightarrow \overline{v_k} \in p^{\mathcal{A}}_{\overline{\tau_{2,j}}}$.
  - Whenever $f_{\overline{\tau_{1,i}} \to \tau_1} \cong_{\mathcal{F}} f_{\overline{\tau_{2,j}} \to \tau_2}$, and $\overline{\tau_k} < \overline{\tau_{1,i}}, \overline{\tau_{2,j}}$ then for all $\overline{v_k} \in \overline{\tau_k}^{\mathcal{A}}$ it is true that $f^{\mathcal{A}}_{\overline{\tau_{1,i}} \to \tau_1}(\overline{v_k}) = f^{\mathcal{A}}_{\overline{\tau_{2,j}} \to \tau_2}(\overline{v_k})$.

  The set of *all algebras over* Σ is denoted by $A(\Sigma)$. The *carrier of* $\mathcal{A}$ is defined by $|\mathcal{A}| := \bigcup_{\tau \in \mathcal{T}} \tau^{\mathcal{A}}$

□

According to this definition, type domains of any two types τ, τ' are disjoint if they have no subtypes in common. This is virtually no limitation and facilitates unambiguous overloading. Notice that we explicitly require constant constructors to have an interpretation. Also, the semantics for the overloaded `inst` predicate is built-in ($\mathtt{inst}^{\mathcal{A}}_{\tau} = \tau^{\mathcal{A}}$). Therefore, we have to prove that this predicate respects the overloading relation. This is a conclusion of the following lemma.

Lemma 3.2.1 (`inst`-predicate well-defined) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$ and types $\tau_k, \tau_1, \tau_2 \in \mathcal{T}$ such that $\tau_k < \tau_1, \tau_2$. Furthermore, given sets $\tau_k^{\mathcal{A}} \subseteq \tau_1^{\mathcal{A}}, \tau_2^{\mathcal{A}}$ and a value $v_k \in \tau_k^{\mathcal{A}}$. Then $v_k \in \tau_1^{\mathcal{A}} \Leftrightarrow v_k \in \tau_2^{\mathcal{A}}$. □

The proof is easy since the element v_k is in both $\tau_1^{\mathcal{A}}$ and $\tau_2^{\mathcal{A}}$. Notice that the required subset relationship in the lemma is assured by the $<$ condition on type domains in Defn. 3.2.3. Since all instance predicates $\mathtt{inst}_{\tau}$ cover the whole type domain $\tau^{\mathcal{A}}$, the lemma is directly applicable. Consequently, instance predicates respect overloading.

Before we introduce the formal syntax and semantics of terms and formulas, we define variable assignments.

Definition 3.2.4 (Variable assignments) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$ and a set X of variables. Then X is suitable for Σ, if

- X is the union of disjoint sets X_τ, $\tau \in \mathcal{T}$,
- X contains non-overloaded symbols, only, i.e., for all τ, τ' it is true that $\tau \neq \tau'$ implies $\{x \mid x : \tau \in X_\tau\} \cap \{x \mid x : \tau' \in X_{\tau'}\} = \emptyset$, and
- X is disjoint to all symbols of Σ.

An element $x : \tau$ is a *qualified variable*, where x is the *name* of that variable and τ is its type. Let X be a set of variables suitable for Σ and $\mathcal{A}$ be a Σ-algebra. A *variable assignment* or *environment* η for X and $\mathcal{A}$ is a function mapping X to $\bigcup_{\tau \in \mathcal{T}} \tau^{\mathcal{A}}$ such that

$$x \in X_\tau \Rightarrow \eta(x) \in \tau^{\mathcal{A}} \cup \{\bot\}$$

Let η be a variable assignment, $x \in X_\tau$ a variable and $v \in \tau^{\mathcal{A}}$. Then the *pointwise update* $\eta[x \mapsto v]$ of η is defined by

$$\eta[x \mapsto v](y) := \begin{cases} v, & x = y \\ \eta(y), & \text{otherwise} \end{cases} .$$

We also call it the *update of* η *at* x. The set $Env(X, \mathcal{A})$ denotes the set of all environments for X and $\mathcal{A}$. Moreover, we denote the set of all environments for X, Σ by $Env(X, \Sigma) := \bigcup_{\mathcal{A} \in A(\Sigma)} Env(X, \mathcal{A})$. □

With these preliminaries we can define the syntax and semantics of terms and formulas over given signatures. Throughout this thesis we will, however, have to define different languages. They mostly differ in the syntax and semantics of their terms and *atomic* formulas, only. Their extensions to formulas of first order logic (FOL formulas) including logical operators like $\wedge$ and $\neg$ and quantifiers are done quite similar. Therefore, we introduce term and formula structures and appropriate extension operators for these structures in order to shorten notation.

Definition 3.2.5 (Term structures, formula structures) Given a signature Σ and a set X of variables suitable for X. Then a *term structure* $TS_{X,\Sigma} := (T, \overset{t}{\rightsquigarrow}, \mathcal{FV})$ *over* Σ, X consists of

(1) a set T of *terms over* X, Σ such that T is the union of terms T_τ of type τ, i.e., $T = \bigcup_{\tau \in \mathcal{T}} T_\tau$, such that $\tau < \tau' \Rightarrow T_\tau \subseteq T_{\tau'}$ for all $\tau, \tau' \in \mathcal{T}$,

(2) a *value relation* $\overset{t}{\rightsquigarrow} \subseteq A(\Sigma) \times Env(X, \Sigma) \times T \times \bigcup_{\mathcal{A} \in A(\Sigma)} |\mathcal{A}| \cup \{\bot\}$ such that
$t \in T_\tau \Rightarrow (\tau^{\mathcal{A}} \cup \{\bot\}) \cap \{v \mid (\mathcal{A}, \eta, t) \overset{t}{\rightsquigarrow} v\} \neq \emptyset$ for all $\mathcal{A}, \eta, t, \tau$.

(3) a mapping $\mathcal{FV} : T \to 2^X$ returning the set of *free variables*[3].

A *formula structure* $FS_{X,\Sigma} := (F, Mod, \models, \mathcal{FV})$ over X, Σ consists of

(1) a set of F of *formulas* over X, Σ,

(2) a set Mod of *models*,

[3] 2^X denotes the power set of X

(3) a *satisfaction relation* $\models \subseteq Mod \times Env(X, \Sigma) \times F$, and
(4) a mapping $\mathcal{FV} : F \to 2^X$ returning the set of *free variables*.

□

The components of term structures include terms in T and their semantics. In particular, the value relation determines values for terms t w.r.t. a given Σ-algebra $\mathcal{A}$ and a variable assignment η. In general $\overset{t}{\rightsquigarrow}$ need not be a function. Also, we do not require the sets T_τ to be disjoint. We will, for example introduce concept terms. There, a given term can have different types and values. Yet we require that t has a value in $\tau^{\mathcal{A}} \cup \{\bot\}$ if t is of type τ.

Similar to term structures, formula structures carry syntax and semantics of formulas. Yet they contain an explicit set of models. The satisfaction relation $\models$ is then determined w.r.t. a model $M \in Mod$, a formula ϕ, and a variable assignment. In contrast to terms, formulas of languages that we introduce in this thesis will sometimes be evaluated w.r.t. models other than Σ-algebras. Preservation requirements, e.g., are evaluated w.r.t. migration sequences.

The following definition introduces Σ-terms as a term structure.

Definition 3.2.6 (Term syntax and semantics) Let Σ be a signature and X a suitable set of variables for Σ. Then *the term structure* $T_{X,\Sigma}$ *for terms over* Σ, X is defined by

$$T_{X,\Sigma} := (T(X,\Sigma), \overset{t}{\rightsquigarrow}, \mathcal{FV}), T(X,\Sigma) = \bigcup_{\tau \in \mathcal{T}} T_\tau(X,\Sigma),$$

where *the sets* $T_\tau(X,\Sigma)$*,* $\tau \in \mathcal{T}$*, of* Σ*-terms of type* τ, the value relation $\overset{t}{\rightsquigarrow}$, the function $\mathcal{FV}$, and the set $T_\tau^{\leq}(X,t)$ of subterms of t of type τ are defined inductively as shown in Tab. 3.2. Terms with $\mathcal{FV}(t) = \emptyset$ are called *ground terms*. We denote the set of ground terms over Σ of type τ by $GT_\tau(\Sigma)$. □

As shown here, we refer to the term *set* by $T(X,\Sigma)$ and to the corresponding term *structure* by $T_{X,\Sigma}$. The rules of the calculus in Tab. 3.2 combine well-formedness and well-typedness of terms. To keep notation short, we will identify term tuples by $\overline{t_i}$ in the following whenever the value of i plays no role or is clear from the context. A function application $f(\overline{t_i})$ then denotes the application of f to $\overline{t_i}$. Apart from that, *syntactic* term identity, which will be denoted by $\equiv$.

Sample derivations for terms and subterms using the calculus in Tab. 3.2 can be found in App. B.1, page 182. According to the following lemma $T_{X,\Sigma}$ is a well-defined term structure (cf. Defn. 3.2.5).

Lemma 3.2.2 (Σ terms are well-defined) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$ and a set X of variables suitable for Σ. Then $T_{X,\Sigma}$ is a well-defined term structure, i.e.,

(1) For all $\tau, \tau' \in \mathcal{T}$ it is true that $\tau < \tau'$ implies $T_\tau(X,\Sigma) \subseteq T_{\tau'}(X,\Sigma)$.
(2) For all types $\tau \in \mathcal{T}$, all terms in $T_\tau(X,\Sigma)$, all algebras $\mathcal{A} \in A(\Sigma)$ and variable assignments $\eta \in Env(X,\mathcal{A})$ there is $v \in \tau^{\mathcal{A}} \cup \{\bot\}$ such that $(\mathcal{A}, \eta, t) \overset{t}{\rightsquigarrow} v$.

□

Table 3.2: *Syntax and semantics of terms*

Syntax:

(1) **Variables:** $\dfrac{x:\tau \in X_\tau}{x \in T_\tau(X,\Sigma)}$ $\quad \mathcal{FV}(x:\tau) = \{x:\tau\}$

(2) **Functions:** $\dfrac{t_1 \in T_{\tau_1}(X,\Sigma), ..., t_n \in T_{\tau_n}(X,\Sigma)\ f_{\overline{\tau_i}\to\tau} \in \mathcal{F}}{f(t_1,...,t_n) \in T_\tau(X,\Sigma)}$ $\quad \mathcal{FV}(f(t_1,...,t_n)) = \bigcup_i \mathcal{FV}(t_i)$

(3) **Subtyping:** $\dfrac{t \in T_{\tau'}(X,\Sigma)\ \tau'<\tau}{t \in T_\tau(X,\Sigma)}$

(4) **Subterms:** $\dfrac{t \in T_\tau(X,\Sigma)}{t \in T^{\leq}_\tau(X,t)}$ $\quad \dfrac{f(t_1,...,t_n) \in T_\tau(X,\Sigma) \quad t_i \in T_{\tau_i}(X,\Sigma)\ (1 \leq i \leq n)}{t_i \in T^{\leq}_{\tau_i}(X,f(t_1,...,t_n))}$ $\quad \dfrac{t \in T^{\leq}_\tau(X,t') \quad t' \in T^{\leq}_{\tau'}(X,t'')}{t \in T^{\leq}_\tau(X,t'')}$

Semantics:

(1) **Variables:** $\dfrac{x:\tau \in X_\tau \quad \eta \in Env(X,\mathcal{A})}{(\mathcal{A},\eta,x) \overset{t}{\rightsquigarrow} \eta(x)}$

(2) **Function application:**

$$\frac{\begin{array}{ll} t_1 \in T_{\tau_1}(X,\Sigma), ..., t_n \in T_{\tau_n}(X,\Sigma) & \mathcal{A} \in A(\Sigma), \eta \in Env(X,\mathcal{A}) \\ f_{\overline{\tau_i}\to\tau} \in \mathcal{F} & (\mathcal{A},\eta,t_1) \overset{t}{\rightsquigarrow} v_1, ..., (\mathcal{A},\eta,t_n) \overset{t}{\rightsquigarrow} v_n \\ & v_1 \in \tau_1^{\mathcal{A}} \cup \{\bot\}, ..., v_n \in \tau_n^{\mathcal{A}} \cup \{\bot\} \end{array}}{(\mathcal{A},\eta,f(t_1,...,t_n)) \overset{t}{\rightsquigarrow} f^{\mathcal{A}^\bot}_{\overline{\tau_i}\to\tau}(v_1,...,v_n)}$$

The proof can be found in App. C.1, page 204, and goes by straightforward induction on t. Generally, the value of a term is not uniquely determined as we permit overloading. In the following lemma we provide a sufficient condition for global uniqueness of derived term values.

Lemma 3.2.3 (Unique term values) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$, a set X of variables suitable for Σ, a Σ-algebra $\mathcal{A} \in A(\Sigma)$, and a variable assignment $\eta \in Env(X,\mathcal{A})$. If there is $\texttt{Top} \in \mathcal{T}$ such that $\tau<\texttt{Top}$ for all $\tau \in \mathcal{T}$, the following holds: For all terms $t \in T(X,\Sigma)$ it is true that $(\mathcal{A},\eta,t) \overset{t}{\rightsquigarrow} v_1$ and $(\mathcal{A},\eta,t) \overset{t}{\rightsquigarrow} v_2$ together imply $v_1 = v_2$. □

The proof is provided in App. C.1, page 204. We will require existence of a type `Top` for the basic DA later on. In this way we assure uniqueness of term values and implement a JAVA-like overloading (cf. JAVA type *Object*). In particular, type `Top` existing and adherence to the overloading relation together exclude the pattern $\texttt{test} : \tau \to \tau_1$ and $\texttt{test} : \tau \to \tau_2$ for types τ_1, τ_2 with disjoint carriers. Functions having equal parameter types and names may not have disjoint return types. In JAVA even equal return types are required. The benefit of this restricted overloading has just been expressed; term values are uniquely determined. Also, type checking is accelerated in practice.

The presence of a greatest type `Top` justifies the introduction of an explicit *value function* $\mathcal{V}[\![\_]\!]$, which is standard in the literature where overloading is not permitted.

Definition 3.2.7 (Value function for terms) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$ such that there is $\texttt{Top} \in \mathcal{T}$ such that $\tau<\texttt{Top}$ for all $\tau \in \tau'$. Given furthermore $\mathcal{A} \in A(\Sigma)$, X suitable for Σ, $\eta \in Env(X,\Sigma)$ and a term $t \in T(X,\Sigma)$. Then

$$\mathcal{V}^{\mathcal{A}}[\![t]\!]\eta = v \Leftrightarrow (\mathcal{A},\eta,t) \overset{t}{\rightsquigarrow} v.$$

□

Since the basic DA will have a greatest type, the value function can be used. Next we introduce constructor terms. They will play an important role when we characterize object content.

Definition 3.2.8 (Constructor terms) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$. Then the *set* $CT_\tau(\Sigma)$ *of constructor terms over* Σ is defined by

$$CT_\tau(\Sigma) := GT_\tau(\Sigma')$$

where $\Sigma' = (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{C})$. Given a type τ and a constructor term $t \in CT_\tau(\Sigma)$ the *length* $l(t)$ *of* t is inductively defined as follows:

- $l(c) = 1$, iff $c \equiv t$ is a constant constructor $c : \tau \in \mathcal{F}_\tau$.
- $l(c(t_1, ..., t_n)) = \sum_{i=1}^{n} l(t_i)$.

□

Σ' rules out all non-constructor functions. All *ground* terms over Σ' are constructor terms for Σ. The length of constructor terms is determined by counting the number of occurrences of constructors. Constructor length is needed in the following definition, where we formalize the notions of minimal constructors and generatedness. The latter is often referred to as the "no junk"-principle in the literature [BM04]. It assures that all elements in a type domain are indeed produced by a constructor term.

Definition 3.2.9 (Generatedness) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$, a type $\tau \in \mathcal{T}$, a Σ-algebra $\mathcal{A}$, and a value $v \in \tau^{\mathcal{A}}$. Then v *is constructor-generated*, iff there is a term t in $CT_\tau(\Sigma)$ such that $(\mathcal{A}, \emptyset, t) \overset{t}{\rightsquigarrow} v$. The set of all constructor terms that generate v is denoted by $CT_{\mathcal{A}}(\Sigma, v)$. Moreover, the *set* $CT^{min}_{\mathcal{A}}(\Sigma, v)$ *of constructor terms of minimal length that generate* v (or *minimal constructor terms for* v*)* is determined by

$$CT^{min}_{\mathcal{A}}(\Sigma, v) := \{t \mid t \in CT_{\mathcal{A}}(\Sigma, v), \forall t' \in CT_{\mathcal{A}}(\Sigma, v) \bullet l(t') \geq l(t)\}$$

□

Later on, we will only permit term-generated specification models. Constructor terms of minimal length will be used to characterize those digital objects that are part of the *content* of another digital object.

Having defined terms and their semantics, we now switch to Σ-formulas. As explained before, we aim at suitable extension operators that can be applied to term and formula structures. Σ-formulas will then be defined in terms of these operators.

Definition 3.2.10 (Extension operators for term / formula structures) Given a term structure $TS_{X,\Sigma} := (T, \overset{t}{\rightsquigarrow}, \mathcal{FV})$, a formula structure $FS_{X,\Sigma} := (F, Mod, \models, \mathcal{FV})$, and a *domain mapping*

$$dom : (Mod \times X \times Env(X, \Sigma)) \rightarrow (A(\Sigma) \times \bigcup_{\mathcal{A} \in A(\Sigma)} 2^{|\mathcal{A}|})$$

such that

$$dom(M, x : \tau, \eta) = (\mathcal{A}, d) \Rightarrow \eta \in Env(X, \mathcal{A}) \wedge d \subseteq \tau^{\mathcal{A}}.$$

Then we define:

(1) $AT(TS_{X,\Sigma}) := (F^{at}(T_{X,\Sigma}), A(\Sigma), \models_t, \mathcal{FV}^{at})$ (atomic formulas over $TS_{X,\Sigma}$)
(2) $FOL(FS_{X,\Sigma}, dom) := (F^{fol}(FS_{X,\Sigma}), Mod, \models_{fol}, \mathcal{FV}^{fol})$ (FOL formulas over $FS_{X,\Sigma}$ under dom)

where the respective components

(1) $F^{at}(TS_{X,\Sigma})$, $\models_t$, $\mathcal{FV}^{at}$, and
(2) $F^{fol}(FS_{X,\Sigma})$, $\models_{fol}$, and $\mathcal{FV}^{fol}$

are determined by the rules shown in Tab. 3.3.

Given a term structure $TS'_{X,\Sigma} := (T', \overset{t'}{\rightsquigarrow}, \mathcal{FV}')$. Then *the join* $TS_{X,\Sigma} \uplus TS'_{X,\Sigma}$ *of* $TS_{X,\Sigma}$ *and* $TS'_{X,\Sigma}$ is defined, iff

$$((\mathcal{A}, \eta, t) \overset{t}{\rightsquigarrow} v \Leftrightarrow (\mathcal{A}, \eta, t) \overset{t'}{\rightsquigarrow} v) \wedge \mathcal{FV}(t) = \mathcal{FV}'(t) \text{ for all } (\mathcal{A}, \eta, t, v) \in A(\Sigma) \times Env(X, \Sigma) \times (T \cap T') \times \bigcup_{\mathcal{A} \in A(\Sigma)} |\mathcal{A}| \cup \{\bot\}$$

Then $TS_{X,\Sigma} \uplus TS'_{X,\Sigma} := (T \cup T', \overset{t}{\rightsquigarrow} \cup \overset{t'}{\rightsquigarrow}, \mathcal{FV} \cup \mathcal{FV}')$. □

On the whole, Defn. 3.2.10 introduces three operators that can be applied to term and formula structures, respectively. The components of the first two operators are depicted in Tab. 3.3. Given a term structure $TS_{X,\Sigma} := (T, \overset{t}{\rightsquigarrow}, \mathcal{FV})$, $AT(TS_{X,\Sigma})$ yields the formula structure for atomic formulas over T. They comprise term equality and predicate application for a given predicate in $\mathcal{P}$. The semantics is standard. Yet notice that we use an existential equality. Both terms of a formula $s = t$ must be defined in order to conclude $s = t$.

The operator $FOL(FS_{X,\Sigma}, dom)$ lifts a given *formula structure* $FS_{X,\Sigma}$ to a structure of first order logic. Rule (1) integrates the original structure $FS_{X,\Sigma}$. Rules (2) and (3) supplement conjunction $\wedge$ and negation $\neg$. In this way, $FOL(FS_{X,\Sigma}, dom)$ covers formulas of propositional logic. The induced semantics is standard. Notice, however, that we use arbitrary models $M \in Mod$ instead of Σ-algebras. Rule (4) adds the universal quantifier $\forall$. Therefore, the FOL operator requires an explicit *domain mapping dom*. Given a formula $\forall x : \tau \bullet \phi$, this mapping determines the quantification scope for $x : \tau$. In particular, the domain is computed w.r.t. a model M, a variable $x : \tau$, and an environment η. In order to assure that the semantics is well-defined, the result $(\mathcal{A}, d)$ of $dom(M, x : \tau, \eta)$ must be suitable for η. In particular, the update $\eta[x \mapsto a]$ must be defined. Also, the type τ of x must be respected. Therefore, we explicitly require

$$dom(M, x : \tau, \eta) = (\mathcal{A}, d) \Rightarrow \eta \in Env(X, \mathcal{A}) \wedge d \subseteq \tau^{\mathcal{A}}$$

in Defn. 3.2.10. Disregarding the genericness coming from the explicit domain mapping, the semantics for the universal quantifier $\forall$ is standard. On the whole, our syntax merely supports a minimal set $\wedge, \neg, \forall$ of logical operators in order to keep proofs short. Yet we will use $\vee$ (or), $\Rightarrow$ (implies), $\Leftarrow$ (follows from), $\Leftrightarrow$ (equivalent to), and $\exists$ (exists) with their usual translations using the operators of Tab. 3.3.

When introducing the preservation language in Chap. 5, we will employ the join-operator $\uplus$. It is used to join two term structures. In order to prevent ambiguity, the

Table 3.3: *Semantics of extension operators for term- and formula structures*

Atomic formulas over $TS_{X,\Sigma}$:

(1) **Equality:** $\dfrac{s \in T_\tau,\ t \in T_{\tau'}\ \ \tau{<}\tau' \vee \tau'{<}\tau}{s = t \in F^{at}(TS_{X,\Sigma})}$ $\quad \mathcal{FV}^{at}(s=t) = \mathcal{FV}(s) \cup \mathcal{FV}(t)$

(2) **Predicates:** $\dfrac{t_1 \in T_{\tau_1}, ..., t_n \in T_{\tau_n} \quad p_{\overline{\tau_i}} \in \mathcal{P}}{p(t_1,...,t_n) \in F^{at}(TS_{X,\Sigma})}$ $\quad \mathcal{FV}^{at}(p(t_1,...,t_n)) = \bigcup_{1 \le i \le n} \mathcal{FV}(t_i)$

---

(1) **Equality:** $\dfrac{s = t \in F^{at}(TS_{X,\Sigma}) \quad \mathcal{A} \in A(\Sigma), \eta \in Env(X,\mathcal{A}) \quad (\mathcal{A},\eta,s) \overset{t}{\rightsquigarrow} v,\ (\mathcal{A},\eta,t) \overset{t}{\rightsquigarrow} v, v \neq \bot}{\mathcal{A} \models_t s = t[\eta]}$

(2) **Predicates:** $\dfrac{\begin{array}{ll} t_1 \in T_{\tau_1}, ..., t_n \in T_{\tau_n} & (\mathcal{A},\eta,t_1) \overset{t}{\rightsquigarrow} v_1, ..., (\mathcal{A},\eta,t_n) \overset{t}{\rightsquigarrow} v_n \\ p_{\overline{\tau_i}} \in \mathcal{P}_{\overline{\tau_i}} & v_1 \in \tau_1^{\mathcal{A}} \cup \{\bot\}, ..., v_n \in \tau_1^{\mathcal{A}} \cup \{\bot\} \\ \mathcal{A} \in A(\Sigma), \eta \in Env(X,\mathcal{A}) & \overline{v_i} \in p^{\mathcal{A}}_{\overline{\tau_i}} \end{array}}{\mathcal{A} \models_t p(t_1,...,t_n)[\eta]}$

FOL-formulas over $FS_{X,\Sigma}$:

(1) **Formulas of $FS_{X,\Sigma}$:** $\dfrac{\phi \in F}{\phi \in F^{fol}(FS_{X,\Sigma})}$ $\quad \mathcal{FV}^{fol}(\phi) = \mathcal{FV}(\phi)$

(2) **Negation:** $\dfrac{\phi \in F^{fol}(FS_{X,\Sigma})}{\neg\phi \in F^{fol}(FS_{X,\Sigma})}$ $\quad \mathcal{FV}^{fol}(\neg\phi) = \mathcal{FV}^{fol}(\phi)$

(3) **Conjunction:** $\dfrac{\phi \in F^{fol}(FS_{X,\Sigma})\ \ \psi \in F^{fol}(FS_{X,\Sigma})}{\phi \wedge \psi \in F^{fol}(FS_{X,\Sigma})}$ $\quad \mathcal{FV}^{fol}(\phi \wedge \psi) = \mathcal{FV}^{fol}(\phi) \cup \mathcal{FV}^{fol}(\psi)$

(4) **Quantification:** $\dfrac{\phi \in F^{fol}(FS_{X,\Sigma})\ \ x : \tau \in X}{\forall x \bullet \phi \in F^{fol}(FS_{X,\Sigma})}$ $\quad \mathcal{FV}^{fol}(\forall x : \tau \bullet \phi) = \mathcal{FV}^{fol}(\phi) \backslash \{x\}_s$

---

(1) **Formulas of $FS_{X,\Sigma}$:** $\dfrac{\phi \in F \quad M \in Mod \quad \eta \in Env(X,\Sigma) \quad M \models \phi[\eta]}{M \models_{fol} \phi[\eta]}$

(2) **Negation:** $\dfrac{\neg\phi \in F^{fol}(FS_{X,\Sigma}) \quad M \in Mod \quad \eta \in Env(X,\Sigma) \quad not\ M \models_{fol} \phi[\eta]}{M \models_{fol} \neg\phi[\eta]}$

(3) **Conjunction:** $\dfrac{\phi \wedge \psi \in F^{fol}(FS_{X,\Sigma}) \quad M \in Mod \quad \eta \in Env(X,\Sigma) \quad M \models_{fol} \phi[\eta] \text{ and } M \models_{fol} \psi[\eta]}{M \models_{fol} \phi \wedge \psi[\eta]}$

(4) **Quantification:**
$$\dfrac{\forall x : \tau \bullet F^{fol}(FS_{X,\Sigma}) \quad dom(M,x,\eta) = (\mathcal{A},d) \quad \eta \in Env(X,\Sigma) \quad M \models_{fol} \phi[\eta[x \mapsto a]] \text{ for all } a \in d}{M \models_{fol} \forall x : \tau \bullet \phi[\eta]}$$

structures must handle all those terms equally that both of them share. In particular, the set of free variables and the satisfaction relations must work equally.

In the following definition we introduce regular Σ-formulas in terms of the just-provided operators.

Definition 3.2.11 (Formula syntax and semantics, sentences) Let Σ be a signature and X a suitable set of variables for Σ. Then *the formula structure $F_{X,\Sigma}$ for formulas over Σ, X* is defined by $F_{X,\Sigma} := FOL(AT(T_{X,\Sigma}), dom)$, where

$$dom(\mathcal{A}, x : \tau, \eta) := \begin{cases} (\mathcal{A}, \tau^{\mathcal{A}}), & \eta \in Env(X,\mathcal{A}) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

for all $(\mathcal{A}, x : \tau, \eta) \in A(\Sigma) \times X \times Env(X,\Sigma)$.
Throughout this thesis, we identify the components of $F_{X,\Sigma}$ by $F(X,\Sigma)$, $\models$, and $\mathcal{FV}$, respectively. We denote the fact that $\mathcal{A} \models \phi[\eta]$ cannot be derived by $\mathcal{A} \not\models \phi[\eta]$. A

formula ϕ is a *sentence over* Σ, if ϕ contains no free variables, i.e., $\mathcal{FV}(\phi) = \emptyset$. The set of all sentences over Σ is denoted by $\mathcal{S}en(\Sigma)$. □

Closed formulas or *sentences* are important in ADT specifications.

Since we use $FOL(,)$ in Defn. 3.2.11, we have to prove that (1) the quantification scopes determined by *dom* contain elements in $\tau^{\mathcal{A}}$ only and (2) η is a suitable environment for $X, \mathcal{A}$ (cf. Defn. 3.2.10). Yet this directly follows from the definition of *dom*. Since we will need it later on, we fix this result in the following corollary.

Corollary 3.2.1 (Σ formulas well-defined) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$, and a set X of variables suitable for Σ. Then $F_{X,\Sigma}$ is a well-defined formula structure, i.e.,

(1) For all $\mathcal{A} \in A(\Sigma)$, $x : \tau \in X$, and $\eta \in Env(X, \Sigma)$ it is true that $dom(\mathcal{A}, x : \tau, \eta) = (\mathcal{A}', d)$ implies $\eta \in Env(X, \mathcal{A}')$ and $d \subseteq \tau^{\mathcal{A}'}$.

□

Specifications, specification models and formal specification extensions are introduced in the following definition.

Definition 3.2.12 (Specifications, models, extensions) A *specification over* Σ is a tuple $Spec := (\Sigma, \mathcal{S}en)$ such that $\mathcal{S}en \subseteq \mathcal{S}en(\Sigma)$. A *model for Spec* is an algebra $\mathcal{A} \in A(\Sigma)$ such that

(1) $\mathcal{A} \models \phi$ for all formulas $\phi \in \mathcal{S}en$ and
(2) for all types τ and elements $v \in \tau^{\mathcal{A}}$ the set of constructor terms is not empty, i.e., $CT_{\mathcal{A}}(\Sigma, v) \neq \emptyset$.

A specification $Spec' := (\Sigma', \mathcal{S}en')$ is an *extension of Spec*, iff

(1) $\Sigma \subseteq \Sigma'$ and
(2) $\mathcal{A}'|_{\Sigma} \models \phi$ for all models $\mathcal{A}'$ for $Spec'$ and all sentences $\phi \in \mathcal{S}en$.

□

Notice that we permit constructor-generated models, only. Specification extensions will be important later on. We provide a specification for a basic DA, only. This specification has to be extended in order to include user-defined datatypes. In the definition above, $\mathcal{A}'|_{\Sigma}$ denotes the *reduct of* $\mathcal{A}'$ *to* Σ.

Notice that the set of all models is usually conceived to be the semantics of a specification. In this thesis we do not go into detail with model theory, but refer the reader to [EGL89, CoF04]. Yet we mention that models may not exist. Therefore, we will always give reasons why models exist or directly provide them for those specifications that are used in this thesis.

3.3 Basic Formal Digital Archive

Having introduced the technical vehicles for ADT specifications, we can introduce DAs. In particular, we will specify a basic system that comprises datatypes for the supported content of DAs, dynamic functions that control the evolution of DAs, and controlled state transitions. We conclude this section by providing a notion of migration algorithms. It covers sequences of basic state changes.

Table 3.4: *Static signature* Σ_s^{DA} *of basic DA*

Types $\mathcal{T}_s^{DA}$**:**	
`Top`	Supertype of all types
`DObj`	Supertype of all object types
`OID`	Object IDs
`Bool`	Boolean values
`Set`$[\alpha]$	Generic type for sets
Predicates $\mathcal{P}_s^{DA}$**:**	
$<_{id}$: `OID` $\times$ `OID`	Strict ordering on object IDs
empty : `Set`$[\alpha]$	Set emptiness
$\in$: $\alpha \times$ `Set`$[\alpha]$	Set containment
$\subseteq$: `Set`$[\alpha] \times$ `Set`$[\alpha]$	Set inclusion
Static functions $\mathcal{F}_s^{DA}$**:**	
`initID` : `OID`	Initial object ID (constructor)
`nextID` : `OID` $\rightarrow$ `OID`	Next object ID (constructor)
`createDObj` : `OID` $\rightarrow$ `DObj`	Constructor for type `DObj`
`oid` : `DObj` $\rightarrow$ `OID`	Attribute `oid` of type `DObj`
`True` : `Bool`, `False` : `Bool`	Boolean values true and false (constructors)
$\{\}$: `Set`	Empty set (constructor)
$\{\}_s : \alpha \rightarrow$ `Set`$[\alpha]$	Singleton set (constructor)
$\cup$: `Set`$[\alpha] \times$ `Set`$[\alpha] \rightarrow$ `Set`$[\alpha]$	Set union (constructor)
$\setminus$: `Set`$[\alpha] \times$ `Set`$[\alpha] \rightarrow$ `Set`$[\alpha]$	Set difference
`rep` : `Set`$[\alpha] \rightarrow \alpha$	Selector for a representative of the set

3.3.1 Static and Dynamic Signature

In the following we will require all specifications that we consider to be an extension of the specification of the basic DA.

Definition 3.3.1 (Static parts of basic Digital Archive) The static specification of the basic DA $Spec_s^{DA} := (\Sigma_s^{DA}, \mathcal{S}en_s^{DA})$ is given by the static signature Σ_s^{DA} shown in Tab. 3.4 and the axioms of the data types specified in App. A.
The set of subtype axioms $<_s^{DA}$ is the least relation over $\mathcal{T}_s^{DA} \times \mathcal{T}_s^{DA}$ satisfying the subtype conditions:

(1) $<_s^{DA}$ is reflexive, transitive, and antisymmetric on $\mathcal{T}_s^{DA}$.
(2) For all $\tau \in \mathcal{T}_s^{DA}$ it is true that $\tau <_s^{DA}$ `Top`.
(3) For all generic types $\tau[\overline{\rho_i}]$ and $\overline{\tau_{1i}}, \overline{\tau_{2i}} \in (\mathcal{T}_s^{DA})^*$ the following holds: $\overline{\tau_{1i} <_s^{DA} \tau_{2i}} \Rightarrow \tau[\overline{\tau_{1i}}] <_s \tau[\overline{\tau_{2i}}]$.

□

In the following we will often omit the indices when referring to the components of a signature if the meaning is clear from the context. The basic DA includes a basic set of types only. The subtype conditions assure that (1) $<_s^{DA}$ is an ordering and (2) there is a greatest type `Top`. Recall that this implies uniqueness of term values (cf. Lemma 3.2.3). Also, we require generic types to preserve subtype relationships. Notice that we distinguish two kinds of objects. Digital objects have an *object type* (subtype of `DObj`). All other elements have a *basic type* (no subtype of `DObj`). There,

we explicitly permit types that are subtypes of basic types and object types at the same time.

All types in $\mathcal{T}_s^{DA}$ are specified formally as abstract data types in App. A, where types `Bool` and `Set`$[\alpha]$ are standard (cf. [CoF04]). Notice that we use a function `rep` returning a *representative* of a given set. It is non-deterministic in general. If applied to a set of digital objects, however, it returns the representatives in ascending order w.r.t. their ID.

Recall that `OID` captures object IDs. We define an initial ID `initID` and proceed to the next ID using `nextID`. We demand injectivity of `nextID` and a strict ordering relation $<_{id}$ on object IDs. In the dynamic system irreflexivity of this relation assures that `nextID` computes an ID which has not been used before.

Lemma 3.3.1 (`OID` circle-free) Given type `OID` as specified in App. A. Then we have

$$\forall id : \texttt{OID}, i \geq 1 \bullet id \neq \texttt{nextID}^i(id).$$

□

The proof can be found in App. C.1. According to this lemma, `OID` is circle-free w.r.t. `nextID`. The natural numbers with $0 = \texttt{initID}$, $(+1) = \texttt{nextID}$ and $< = <_{id}$ are an example model for `OID`.

As explained before, the record type `DObj` is the supertype of all digital objects stored in the DA. Later on, we control object histories of objects of this type and its subtypes, only. The specification comprises a constructor `createDObj` and an attribute `oid`.

The basic DA of Defn. 3.3.1 can be extended in a controlled manner.

Definition 3.3.2 (Permissible extensions) A specification $Spec := (\Sigma, \mathcal{S}en)$ is a *permissible extension of* $Spec_s^{DA}$, iff $Spec$ is an extension of $Spec_s^{DA}$, $<$ satisfies the subtype conditions, all types τ<`DObj` satisfy the *ID-property*

(1) All constructors for τ contain exactly one variable of type `OID` in the first position.
(2) The property $\forall id : \texttt{OID}, \overline{x_i : \tau_i} \bullet \texttt{oid}(\texttt{c}(id, \overline{x_i})) = id$ holds for all constructors $\texttt{c} : \texttt{OID} \times \overline{\tau_i} \to \tau$ of τ.

and the *constructor property*

(1) For all terms t, t' in $GT_{\texttt{DObj}}(\Sigma)$ it is true that whenever $\mathcal{A} \models t = t'$ then there is a term $c(\overline{t_i}) \in T^{\leq}_{\texttt{DObj}}(\emptyset, t)$ such that c is a constructor and $\mathcal{A} \models c(\overline{t_i}) = t'$.
(2) For all types $\tau \in \mathcal{T}$ and terms t in $GT_\tau(\Sigma)$ it is true that

$$\{\mathcal{V}^{\mathcal{A}}[\![t_{\texttt{DObj}}]\!] \mid t_{\texttt{DObj}} \in T^{\leq}_{\texttt{DObj}}(\emptyset, t_{min})\} = \{\mathcal{V}^{\mathcal{A}}[\![t_{\texttt{DObj}}]\!] \mid t_{\texttt{DObj}} \in T^{\leq}_{\texttt{DObj}}(\emptyset, t'_{min})\}$$

for all $t_{min}, t'_{min} \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!])$.

□

The notion of a *permissible extension* covers both, introducing new types and extending existing ones as long as the subtype conditions of Defn. 3.3.1, the ID-property, and the constructor-property are obeyed. In particular, all type constructors of all subtypes of `DObj` must have an attribute of type `OID` (ID-property). That way, we can attach IDs on creation.

Notice that both parts of the constructor property are expressed by means of constructor terms:

(1) If t equals a digital object t', t' is included in t already (t' equals a constructor term which is a subterm of t).
(2) Any two minimal constructor terms that construct the same term t contain equal subterms of type `DObj`.

This reflects our intention to characterize the content of digital objects by means of minimal constructor terms later on. On the whole, the constructor-property assures two things:

(1) Functions cannot create new digital objects; this can be done by basic state changes later on. In particular, *digital objects* cannot themselves cause state changes (as opposed to models of object-oriented systems). This is standard for objects of our application domain. They are stored permanently and methods defined on them serve to access their content, only.
(2) Objects that appear as the *content* of other objects already exist on creation of their parent object. Also, they may not be deleted as long as the parent object still exists.

In App. B.1 we explain the constructor and ID property in more detail using some examples.

According to the following lemma, Σ_s^{DA} satisfies both the ID-property and the constructor property and, hence, is a permissible extension of itself.

Lemma 3.3.2 (Static basic DA well-defined) The specification $Spec_s^{DA}$ of the basic DA in Defn. 3.3.1 satisfies the ID-property and the constructor property. □

The proof can be found in App. C.1. Recall that we require all specification models (thus, all models for $Spec_s^{DA}$) to be *term-generated*. This induces a bi-directional correspondence between semantic values and syntactic terms, which is important in our setting. The implementation of our system, e.g., uses JAVA class instances as interpretations. Users can implement transformations in JAVA and construct new class instances. When tracing transformations, our system constructs syntactic representatives of these values. This is possible only in presence of global term-generatedness (which is true for JAVA objects). Notice that non-empty models exist for $Spec_s^{DA}$ since we have specified all data types up to isomorphism and equipped some of them with "initial constructors" (like `initID`). They may not be partial and, hence, must have an interpretation.

Recall that we want to characterize those digital objects that belong to the content of another digital object. This prevents parent objects from unintentional changes to their contents. In the introductory survey to this chapter we, e.g., could not delete the directory "overview" as it was a sub-directory of "source". The following definition introduces *digital objects* and *content of digital objects*.

Definition 3.3.3 (Digital objects, contents) Given a specification $Spec := (\Sigma, Sen)$ that is a permissible extension of $Spec_s^{DA}$ and a Σ-model $\mathcal{A}$ for $Spec$. Then a term $t \in GT_{\mathtt{DObj}}(\Sigma)$ is called a *digital object*.
Given a type τ, a set X of variables suitable for Σ, a term $t \in T_\tau(X, \Sigma)$ and a variable assignment η. Then the *object-valued content of t in $\mathcal{A}$ under η* is defined by

$$cont_{\mathtt{DObj}}^{\mathcal{A},\eta}(t) := T_{\mathtt{DObj}}^{\leq}(\emptyset, ct_t),$$

where ct_t is a fixed minimal constructor term in $CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta)$.
Given a term $t' \in T_{\tau'}(X, \Sigma)$, we, furthermore, define:

$$\mathcal{A} \models t' \in cont^{\mathcal{A},\eta}_{\texttt{DObj}}(t)[\eta] \Leftrightarrow \text{ there is a term } t'' \in cont^{\mathcal{A},\eta}_{\texttt{DObj}}(t) \text{ such that } \mathcal{A} \models t' = t''[\eta].$$

□

The object-valued content of a term t yields the subterms of type `DObj` of an arbitrary, but fixed constructor term of minimal length for t. These terms can be understood as *representatives* of those digital objects that are in the content of t. This "technical" characterization of the object-valued content assures a solid mathematical foundation. In practice, mostly constructor terms of minimal length are used. As an example, the record-like types that we use have one constructor only. Constructor terms of this kind are, thus, always minimal. If a datatype has more than one constructor, but all constructors have disjoint domains, all constructor terms of this type are also minimal. This applies to trees, for example.

Due to part (2) of the constructor property in Defn. 3.3.2, the choice of the constructor term c_t is irrelevant as any two minimal constructor terms that construct equal terms contain the same sub-objects. The following lemma justifies the term "object-valued *content*".

Lemma 3.3.3 (Properties of the relation $cont_{\texttt{DObj}}()$) Given a specification $Spec := (\Sigma, Sen)$, $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$, that is a permissible extension of the basic DA and a Σ-model $\mathcal{A}$ for *Spec*. Then the following holds:

(1) Given terms $s, t \in T_\tau(X, \Sigma)$ and a variable assignment η. Then

$$\mathcal{A} \models s \in cont^{\mathcal{A},\eta}_{\texttt{DObj}}(t)$$
$$\Leftrightarrow$$
$$\text{There is } ct_t \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \text{ such that } \mathcal{V}^{\mathcal{A}}[\![s]\!]\eta \in \{\mathcal{V}^{\mathcal{A}}[\![t']\!]\eta \mid t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct_t)\}.$$

(2) The relation $R^{cont}_{\mathcal{A}} := \{(t, t') \mid t, t' \in GT_{\texttt{DObj}}(\Sigma), \mathcal{A} \models t \in cont^{\mathcal{A}}_{\texttt{DObj}}(t')\}$ is an ordering. In particular:

(2.1) $R^{cont}_{\mathcal{A}}$ is reflexive, i.e., $\forall t \in GT_{\texttt{DObj}}(\Sigma) \bullet (t, t) \in R^{cont}_{\mathcal{A}}$.

(2.2) $R^{cont}_{\mathcal{A}}$ is transitive, i.e., $\forall s, t, u \in GT_{\texttt{DObj}}(\Sigma) \bullet (s, t) \in R^{cont}_{\mathcal{A}} \wedge (t, u) \in R^{cont}_{\mathcal{A}} \Rightarrow (s, u) \in R^{cont}_{\mathcal{A}}$.

(2.3) $R^{cont}_{\mathcal{A}}$ is antisymmetric, i.e., $\forall s, t \in \texttt{DObj}^{\mathcal{A}} \bullet (s, t) \in R^{cont}_{\mathcal{A}} \wedge (t, s) \in R^{cont}_{\mathcal{A}} \Rightarrow \mathcal{A} \models s = t$.

(3) Digital objects that are the results of applying a non-constructor function appear in the object-valued content of one of the arguments of this function application, i.e.:
Given a function $f : \tau_1 \times ... \times \tau_n \to \tau$ such that $\tau <$ `DObj` and $f_{\overline{\tau_i} \to \tau} \notin \mathcal{C}_\tau$, and terms $t_1 \in GT_{\tau_1}(\Sigma), ..., t_n \in GT_{\tau_n}(\Sigma), t \in GT_\tau(\Sigma)$ such that $\mathcal{A} \models f(t_1, ...t_n) = t$. Then there is a t_i $(1 \leq i \leq n)$ such that $\mathcal{A} \models t \in cont^{\mathcal{A}}_{\texttt{DObj}}(t_i)$.

□

The proof is provided in App. C.1, page 206. Property (1) provides an alternative characterization of the object-valued content. Also, it underlines that the definition of $cont_{\texttt{DObj}}()$ does not depend on the choice of the minimal constructor term ct_t in Defn. 3.3.3; we merely demand an arbitrary constructor to exist such that the term value of one of its sub-terms equals the term value of s.

According to property (2), $cont^{\mathcal{A},\eta}_{\texttt{DObj}}(t)$ induces a *semantic* term ordering on the terms of type `DObj`. This is what we want — relate syntactic term construction to a semantic notion of object content.

Property (3) states that all objects that can be extracted from a term are in the object-valued content of that term. In our example, attributes like `home` or `srcDir` extract sub-objects of given objects. Examples can be found in App. B.1.

In our setting we need to a *dynamic* environment and have to distinguish existing objects and non-existing objects. For this purpose we could, e.g., let the type domains evolve; those objects that are in a type domain exist, the others do not. Recall that we implement our basic state changes as ASM programs later on. In standard ASMs, type domains are represented by *predicates* that can be extended by choosing elements from a *reserve non-deterministically*. Baumeister and Zamulin in [BZ00] extend this approach to CASL specifications. In our setting, however, concrete objects are created. The set of existing objects is always a subset of the type domain of type `DObj`. Also, our type domains cannot evolve. They are fixed as models of the basic DA specification. Therefore, we follow [Zam97, Zam98] and model evolving parts by appropriate dynamic functions.

Definition 3.3.4 (Dynamic signature, static and dynamic part) Given a static signature $\Sigma_s := (\mathcal{T}_s, <_s, \mathcal{P}_s, \mathcal{C}_s, \mathcal{F}_s)$ and a Σ_s - state $\mathcal{A}$. A *dynamic signature* $\Sigma_d := (\mathcal{T}_s, <_s, \mathcal{P}_s, \emptyset, \mathcal{F}_d)$ suitable for Σ_s extends Σ_s by a set of *dynamic (evolving) function symbols* $\mathcal{F}_d$ that are typed over $\mathcal{T}_s$ such that $\mathcal{F}_d \cap \mathcal{F}_s = \emptyset$. An *expanded state* for $\Sigma := \Sigma_s \cup \Sigma_d$ additionally contains interpretations $f^{\mathcal{A}}$ for all function symbols $f \in \mathcal{F}_d$. Terms and formulas over Σ are defined similar to Defn. 3.2.11 but may include function symbols of $\mathcal{F}_d$ as well. Analogously, term values and formula semantics expand to the dynamic functions. We identify the *static and dynamic part of* Σ, respectively, by $\Sigma_s = sp(\Sigma)$ and $\Sigma_d = dp(\Sigma)$. □

Notice that constructors always belong to the static part of a signature according to this definition. In the following we introduce the dynamic signature of the basic DA.

Definition 3.3.5 (Dynamic basic Digital Archive) Given the static signature Σ^{DA}_s of our basic DA. The dynamic signature $\Sigma^{DA}_d := (\mathcal{T}^{DA}_s, <^{DA}_s, \mathcal{P}^{DA}_s, \emptyset, \mathcal{F}^{DA}_d)$ for Σ^{DA}_s is determined by

$$\mathcal{F}^{DA}_d := \{\texttt{existDObj} : \texttt{Set}[\texttt{DObj}], \texttt{usedOIDs} : \texttt{Set}[\texttt{OID}]\}.$$

□

Notice that Σ^{DA}_d is suitable for Σ^{DA}_s according to Defn. 3.3.4. The full basic DA including all static and dynamic parts is defined as follows.

Definition 3.3.6 (Basic DA, object existence) The *basic Digital Archive* is given by $Spec^{DA} := (\Sigma^{DA}, Sen_s^{DA}))$, where, $sp(\Sigma^{DA}) = \Sigma_s^{DA}$ and $dp(\Sigma^{DA}) = \Sigma_d^{DA}$. A specification $Spec := (\Sigma, Sen)$ is a *permissible extension of* $Spec^{DA}$, iff $Sen = Sen_{|sp(\Sigma)}$, $(sp(\Sigma), Sen)$ is a permissible extension of $(\Sigma_s^{DA}, Sen_s^{DA})$, and $\Sigma_d^{DA} \subseteq dp(\Sigma)$.
Given a specification $Spec := (\Sigma, Sen)$ that is a permissible extension of $Spec^{DA}$. Then *an object* $t \in GT_{\texttt{DObj}}(\Sigma)$ *exists in a* Σ*-model* $\mathcal{A}$ *of* $Spec$, iff $\mathcal{A} \models t \in \texttt{existDObj}$. □

Permissible extensions of the full basic DA may not contain sentences over dynamic parts. This might prevent state changes, which we introduce in the next section, from being well-defined. Hence, we explicitly require $Sen = Sen_{|sp(\Sigma)}$. Also, this definition assures $\Sigma_s^{DA} \subseteq sp(\Sigma)$ and $\Sigma_d^{DA} \subseteq dp(\Sigma)$ such that we can use both the static and dynamic components of the basic DA whenever we deal with permissible extensions. The notion of object existence in Defn. 3.3.6 is bound to `existDObj`. Together with the set of used IDs (`usedOIDs`), `existDObj` is adjusted on object creation and object deletion.

We conclude this section by summing up some important points:

- The basic DA can be extended in a controlled manner. There, we require the ID-property and the constructor-property to hold.
- Models have to be term-generated.
- Terms of type `DObj` are referred to as digital objects. Digital objects that are part of the content of another digital object are characterized via minimal constructor terms for their parent object; they occur as sub-terms of these constructor terms. This definition is well-defined only because the constructor property requires all minimal constructor terms to contain equal sets of sub-terms of type `DObj`.
- The object-valued content of a term is denoted by $cont_{\texttt{DObj}}^{\mathcal{A},\eta}(t)$. This relation is an ordering.
- We have introduced the dynamic parts of our basic DA. There, we have fixed a notion of object existence. An object exists if it is in `existDObj`.

In the next section we introduce basic state changes and migrations formally. That way we add system dynamics.

3.3.2 Basic State Changes, Migration Sequences

Various kinds of state transition systems and methods for their high-level analysis can be found in the literature. As we aim at *tracing* transformation processes *operationally* and, hence, want to evaluate adherence to pre-defined preservation requirements w.r.t. their input and output and given source and result states only, we do not introduce a formal state transition system. Yet we mention that our approach naturally *induces* a state transition system, where algebras are system states and the basic state changes are transitions. In particular, we characterize all basic operations via pre- and post conditions, only. Since algebras carry semantics and formula validity can be checked w.r.t. given states, this framework can be translated to *Kripke Structures*. Beyond others, Kripke Structures are used for model checking complex systems w.r.t. properties that are formulated in modal or temporal logics [MOSS99]. This may serve as a starting point for future research.

Table 3.5: *Syntax of basic state changes w.r.t. Σ and a Σ-state $\mathcal{A}$*

Syntax:

(1) **Object creation:** $$\frac{t \in \bigcup_{v \in \tau^{\mathcal{A}}} CT^{min}_{\mathcal{A}}(\Sigma, v) \;\; \tau{<}\texttt{DObj}}{\mathsf{cre}(t) \in \mathcal{BT}r_{\tau}(\Sigma, \mathcal{A})}$$

(2) **Object deletion** $$\frac{t \in \bigcup_{v \in \tau^{\mathcal{A}}} CT^{min}_{\mathcal{A}}(\Sigma, v) \;\; \tau{<}\texttt{DObj}}{\mathsf{del}(t) \in \mathcal{BT}r_{\tau}(\Sigma, \mathcal{A})}$$

(3) **Object transformation:** $$\frac{t_{src} \in \bigcup_{v \in \tau^{\mathcal{A}}} CT^{min}_{\mathcal{A}}(\Sigma, v) \;\; \tau{<}\texttt{DObj} \quad t_{trg} \in \bigcup_{v \in \tau'^{\mathcal{A}}} CT^{min}_{\mathcal{A}}(\Sigma, v) \;\; \tau'{<}\texttt{DObj}}{\mathsf{trans}(t_{src} \mapsto t_{trg}) \in \mathcal{BT}r_{\tau'}(\Sigma, \mathcal{A})}$$

The basic DA evolves according to the state transitions

(1) $\mathsf{cre}(t)$,
(2) $\mathsf{trans}(t_{src} \mapsto t_{trg})$, and
(3) $\mathsf{del}(t)$,

where t, t_{src}, t_{trg} are terms of type `DObj`. Object creation introduces t as a new object, transformation produces a new version t_{trg} of t_{src}, and deletion removes t from the system.

In the introduction to this chapter, we have already expressed expected pre-and post conditions. In the next definition we define them formally.

Definition 3.3.7 (Basic state changes) Given a specification $Spec := (\Sigma, Sen)$ such that $Spec$ is a permissible extension of the basic DA. Given a Σ-model $\mathcal{A}$ for $Spec$, the set of *basic state transitions* over Σ and $\mathcal{A}$ is defined by the rules in Tab. 3.5.
A Σ-state $\mathcal{A}'$ *is subsequent to* $\mathcal{A}$ *w.r.t.* $op \in \mathcal{BT}r_{\texttt{DObj}}(\Sigma, \mathcal{A})$, iff $(\mathcal{A}, op) \overset{btr}{\rightsquigarrow} \mathcal{A}'$, where $\overset{btr}{\rightsquigarrow}$ is defined as follows:

- $(\mathcal{A}, \mathsf{cre}(t)) \overset{btr}{\rightsquigarrow} \mathcal{A}'$, iff the following holds:
  (1) $\mathcal{A} \models t' \in \texttt{existDObj}$ for all $t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, t) \backslash \{t\}$.
  (2) $\mathcal{A}' \models t \in \texttt{existDObj} \wedge \texttt{oid}(t) \in \texttt{usedOIDs}$.
  (3) For all objects $t' \in GT_{\texttt{DObj}}(sp(\Sigma))$ it is true that $\mathcal{A} \models t' \in \texttt{existDObj}$ iff $\mathcal{A}' \models t \neq t' \wedge t' \in \texttt{existDObj}$.
  (4) For all terms id of type `OID` it is true that $\mathcal{A} \models id \in \texttt{usedOIDs}$ iff $\mathcal{A}' \models id \neq \texttt{oid}(t) \wedge id \in \texttt{usedOIDs}$.
  (5) $\mathcal{A}_{|sp(\Sigma)} = \mathcal{A}'_{|sp(\Sigma)}$.
- $(\mathcal{A}, \mathsf{del}(t)) \overset{btr}{\rightsquigarrow} \mathcal{A}'$, iff the following holds:
  (1) For all $t' \in GT_{\texttt{DObj}}(sp(\Sigma))$ is true that $\mathcal{A} \models t' \in \texttt{existDObj} \Rightarrow t \notin cont^{\mathcal{A}}_{\texttt{DObj}}(t') \vee t = t'$.
  (2) $\mathcal{A} \models t \in \texttt{existDObj}$.
  (3) For all $t' \in GT_{\texttt{DObj}}(sp(\Sigma))$ it is true that $\mathcal{A} \models t' \in \texttt{existDObj} \wedge t' \neq t$ iff $\mathcal{A}' \models t' \in \texttt{existDObj}$.

(4) For all terms id of type `OID` it is true that $\mathcal{A} \models id \in \texttt{usedOIDs}$ iff $\mathcal{A}' \models id \in \texttt{usedOIDs}$.

(5) $\mathcal{A}_{|sp(\Sigma)} = \mathcal{A}'_{|sp(\Sigma)}$.

- $(\mathcal{A}, \mathsf{trans}(t_{src} \mapsto t_{trg})) \overset{btr}{\rightsquigarrow} \mathcal{A}'$, iff the following holds:

(1) $(\mathcal{A}, \mathsf{cre}(t_{trg})) \overset{btr}{\rightsquigarrow} \mathcal{A}'$.

(2) $\mathcal{A} \models t_{src} \in \texttt{existDObj}$.

□

We restrict the syntax of the basic transitions in $\mathcal{BT}r_{\texttt{DObj}}(\Sigma, \mathcal{A})$ to *constructor terms of minimal length*. Hence, we need a reference algebra $\mathcal{A}$ in order to determine these terms. This speeds up the evaluation of the object-valued content. Recall that these terms contain exactly those digital objects as sub-terms that are in their object-valued content. Also, minimal constructor terms can be evaluated faster in practice, which supports efficient concept matching. Later on, the basic state changes will have to be implemented anyway (they are specified by pre- and post conditions only). This includes generating *ground* terms since we do not store variable assignments in the system. Often, generating constructor terms of minimal length does not cause an overhead in practice. Recall that constructor terms are always minimal w.r.t. a given datatype if (1) only one constructor exists for that type or (2) any two different constructors for that type have disjoint domains. In this thesis, all user-defined datatypes will satisfy this property. In Sect. 6.2 we will implement the basic state changes by corresponding basic operations. These operations are defined in terms of ASM programs and may serve as a reference.

The *subsequence relation* $\overset{btr}{\rightsquigarrow}$ respects the desired pre- and post conditions. Concerning object creation, the requirements assure the following:

(1) All sub-objects of t of type `DObj` already exist in `existDObj`; objects have to be constructed bottom-up. That way we avoid "dead" references. In the example trace in Tab. 3.1, page 30, creating all sub-directories before creating the "source" directory is, thus, mandatory.

(2) t must exist in the target state and its ID has to be registered in the system. This is the intended effect of object creation.

(3) Object creation causes a minimal change to `existDObj`. This includes that all those objects that exist in $\mathcal{A}$ do still exist in $\mathcal{A}'$. Also, we can derive $\mathcal{A} \models t \notin \texttt{existDObj}$ and $\mathcal{A} \models \texttt{oid}(t) \notin \texttt{existDObj}$ (cf. Lemma 3.3.4 below).

(4) Object creation causes a minimal change to `usedOIDs`. Also, t must not have an ID that has already been used in the system. This assures that objects are globally uniquely identifiable.

(5) The static parts of the source algebra $\mathcal{A}$ do not change. In particular type domains are equal in both algebras. Since constructor terms belong to the static parts of a signature, $\mathcal{BT}r_{\texttt{DObj}}(\Sigma, \mathcal{A}) = \mathcal{BT}r_{\texttt{DObj}}(\Sigma, \mathcal{A}')$ if $\mathcal{A}'$ is the result of applying an object creation to $\mathcal{A}$.

Object deletion is inverse to object creation. Hence, an object cannot be deleted if it is a part of another object in `existDObj` (requirement (1)). Recall, e.g., that the

directory "overview" could not be deleted in the example trace in Tab. 3.1. Analogous to object creation, requirements (2) to (5) assure that exactly the respective object is deleted and no further changes are made to the system.

Object transformation works similar to object creation in terms of creating the transformation result t_{trg}. We, however, explicitly require the transformation source t_{src} to exist. This is the intended meaning of transformation: Take an existing object and relate it to a newly created object.

The following lemma lists some elementary properties of the basic state changes.

Lemma 3.3.4 (Properties of basic state changes) Given a specification $Spec := (\Sigma, Sen)$ such that $Spec$ is a permissible extension of the basic DA. Given furthermore Σ-models $\mathcal{A}, \mathcal{A}'$ for $Spec$ and a basic state transition $tr \in \mathcal{BT}r_{\texttt{DObj}}(\Sigma)$ such that $(\mathcal{A}, tr) \overset{btr}{\rightsquigarrow} \mathcal{A}'$. Then the following properties hold:

- If $tr \equiv \mathsf{cre}(t)$ then
  (1) $\mathcal{A} \models t \notin \texttt{existDObj}$.
  (2) $\mathcal{A} \models \texttt{oid}(t) \notin \texttt{usedOIDs}$.
- If $tr \equiv \mathsf{del}(t)$ then
  (3) $\mathcal{A}' \models t \notin \texttt{existDObj}$.

□

The proof can be found in App. C.1, page 208. The *non-operational* specification of the basic transitions using pre-and post conditions has an important advantage: Users can provide their own *implementations*. These implementations are not bound to a specific language. Moreover, their way of function is constrained by the above requirements, only. As an example, the basic DA could be extended by an explicit object history. This would lead to additional requirements concerning the basic state changes. In particular, a state that is the result of a transformation $\mathsf{trans}(t_{src} \mapsto t_{trg})$ should contain a history entry for t_{src} and t_{trg}. We support all user-defined extensions and implementations that refine the specifications related to the basic DA and the basic state changes.

The basic state changes are single-step transitions. Yet we want to capture migration *processes*. For this purpose, we introduce *migration sequences*.

Definition 3.3.8 (Migration sequence) Given a specification $Spec := (\Sigma, Sen)$ such that $Spec$ is a permissible extension of the basic DA. Given furthermore $n \in \mathbb{N}$, Σ-states $\mathcal{A}_0, ..., \mathcal{A}_n$ and operations $tr_1, ..., tr_n \in \mathcal{BT}r_{\texttt{DObj}}(\Sigma)$ such that $(\mathcal{A}_{i-1}, tr_i) \overset{btr}{\rightsquigarrow} \mathcal{A}_i$ for all $1 \leq i \leq n$. Then

$$\Delta := \langle \mathcal{A}_0, tr_1, \mathcal{A}_1, tr_2, ..., \mathcal{A}_{n-1}, tr_n, \mathcal{A}_n \rangle$$

is a *migration sequence over* Σ *of length* n *from* $\mathcal{A}_0$ *to* $\mathcal{A}_n$. We call $\mathcal{A}_0$ the *source state of* Δ, denoted by $src(\Delta)$ and $\mathcal{A}_n$ the *result state of* Δ, denoted by $res(\Delta)$.
Given another migration sequence

$$\Delta' := \langle \mathcal{A}'_0, tr'_1, \mathcal{A}'_1, tr'_2, ..., \mathcal{A}'_{m-1}, tr'_m, \mathcal{A}'_m \rangle$$

of length m from $\mathcal{A}'_0$ to $\mathcal{A}'_m$. Then Δ and Δ' are *composable*, iff $\mathcal{A}_n = \mathcal{A}'_0$. The *composition of* Δ *and* Δ' yields a migration sequence

$$\Delta; \Delta' := \langle \mathcal{A}_0, tr_1, \mathcal{A}_1, tr_1, ..., \mathcal{A}_{n-1}, tr_n, \mathcal{A}'_0, tr'_1, \mathcal{A}'_1, tr'_2, ..., \mathcal{A}'_{m-1}, tr'_m, \mathcal{A}'_m \rangle$$

of length $m + n$ from $\mathcal{A}_0$ to $\mathcal{A}'_m$. The set of all migration sequences over Σ is denoted by $MS(\Sigma)$. □

Defining migration processes as *sequences* of basic state changes may seem simplistic. It is, however, sufficient for our purposes. We leave further examination on parallelism or control features up to future research.

Migration processes can be understood as runs through an abstract state transition system. The ; operation implements regular sequential composition. $\Delta; \Delta'$ is defined only, if the *result state* of Δ equals the *source state* of Δ'. This assures that all basic state changes in $\Delta; \Delta'$ are valid w.r.t. $\overset{btr}{\rightsquigarrow}$. In App. B.1, page 184, we show a sample migration sequence related to the trace of Tab. 3.1 on page 30.

In the following lemma we list some important properties that are *preserved* by the basic state changes.

Lemma 3.3.5 (Properties preserved by basic state changes) Given a specification $Spec := (\Sigma, Sen)$ such that $Spec$ is a permissible extension of the basic DA. Given furthermore Σ-models $\mathcal{A}, \mathcal{A}'$ for $Spec$ and a basic state transition $tr \in \mathcal{BT}r_{\texttt{DObj}}(\Sigma)$ such that $(\mathcal{A}, tr) \overset{btr}{\rightsquigarrow} \mathcal{A}'$. Then the following properties are preserved (if they hold before executing the state change, they hold afterwards as well):

(1) The IDs of all existing objects are registered as used IDs, i.e.,
$\forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow \texttt{oid}(x) \in \texttt{usedOIDs}$

(2) All existing objects have different IDs, i.e.,
$\forall x, x' : \texttt{DObj} \bullet (x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj}) \Rightarrow (\texttt{oid}(x) = \texttt{oid}(x') \Rightarrow x = x')$

(3) The object-valued content of all existing objects does also exist, i.e.,
$\forall x, x' : \texttt{DObj} \bullet x \in \texttt{existDObj} \wedge x' \in cont^{\mathcal{A}}_{\texttt{DObj}}(x) \Rightarrow x' \in \texttt{existDObj}$

(4) Given a constructor term t_{id} of type `OID`, this ID has been used before and no object in `existDObj` has this ID, i.e.,
$t_{id} \in \texttt{usedOIDs} \wedge \forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow \texttt{oid}(x) \neq t_{id}$

□

The proof can be found in App. C.1, page 208. The first property assures that the IDs of all existing objects are registered. According to property (2), existing objects having the same ID have equal values as well. In this way objects are uniquely identifiable. Property (3) states that the object-valued content of all existing digital objects also exists. This is particularly important in order to avoid dead references. Object-valued content cannot be deleted as long as the parent object exists. Preservation of property (4) guarantees that an object's ID can never be used again if the object is deleted. Later on we shall see that these properties imply object immutability.

In the next definition we introduce initial states and a notion of "permissible" derivations using basic state transitions.

Definition 3.3.9 (Initial and derived states) Given a specification $Spec := (\Sigma, Sen)$ such that $Spec$ is a permissible extension of the basic DA. A Σ-model $\mathcal{A}^I$ *is an initial state*, iff $\mathcal{A} \models \texttt{existDObj} = \{\}$. A Σ-model $\mathcal{A}$ *is derived by basic transitions*, iff there is an initial state $\mathcal{A}^I$ and a migration Δ that leads from $\mathcal{A}^I$ to $\mathcal{A}$. □

Initial states do not contain existing objects. A state that is derived by basic transitions must be derived form an initial state via a valid migration sequence. With these prerequisites properties (1) to (3) of Lemma 3.3.5 become system invariants.

Corollary 3.3.1 (System invariants) Given a specification $Spec := (\Sigma, Sen)$ such that *Spec* is a permissible extension of the basic DA. Then properties (1) to (3) of Lemma 3.3.5 *hold* in all Σ-models $\mathcal{A}$ of *Spec* that are derived by basic transitions, i.e.,

(1) $\mathcal{A} \models \forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow \texttt{oid}(x) \in \texttt{usedOIDs}$

(2) $\mathcal{A} \models \forall x, x' : \texttt{DObj} \bullet (x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj}) \Rightarrow (\texttt{oid}(x) = \texttt{oid}(x') \Rightarrow x = x')$, and

(3) $\mathcal{A} \models \forall x, x' : \texttt{DObj} \bullet x \in \texttt{existDObj} \wedge x' \in cont^{\mathcal{A}}_{\texttt{DObj}}(x) \Rightarrow x' \in \texttt{existDObj}$.

□

Property (4) of Lemma 3.3.5 does not necessarily hold in all states. However, when an object with ID *id* is deleted, the property holds henceforth in all subsequent states.

This concludes the introduction of the evolving DA. The basic state changes correspond to a state transition system. Migration sequences are particular runs through this system. They can be used for syntactic tracing of more sophisticated migration processes. These processes, in turn, can be implemented using the functional language that will be introduced in Chap. 6.

3.4 Summary

We have introduced some basic prerequisites for the rest of this thesis. The content of digital objects is modeled using algebraic datatypes. They are particularly suitable in our setting since inductive data structures like trees or graphs frequently occur; this has been underlined by our the running example.

Our basic DA includes some administrative structures like a type `OID` modeling object IDs or a type `DObj` for digital objects. We have introduced conditions for well-founded extensions of the basic DA. They guarantee existence of a suitable notion of *object-valued content*. This is necessary to prevent the content of digital objects from being changed unintentionally.

Apart from these static parts, some dynamic parts have been introduced. The central part here is the function `existDObj`. It stores the set of all existing objects and evolves according to the three state changes (1) object creation, (2) object deletion, and (3) object transformation.

In summary, our dynamic system implements the following properties:

- Object types, functions, and semantic relationships can be described formally as long as they adhere to the following:
  - All objects are identifiable by an ID of type `OID`.
  - An objects ID is considered to be part of its content and can be attached to an object on creation.

    - Functions do not create new digital objects. If a function returns a digital object, this object is part of the object-valued content of one of the arguments of the function.

- Objects are immutable. All IDs are used once only and are attached to objects on creation or transformation.
- An object may not be created unless all its object-valued content exists in the system.
- An object may not be deleted whenever it still has a parent object.

We proceed by introducing the formal syntax and semantics of concept definitions. Concepts are the basis for formal preservation requirements.

Chapter 4

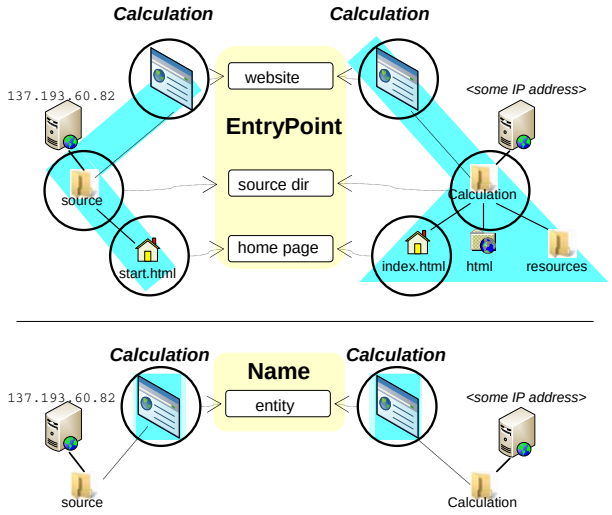
Contexts and Concepts

Concepts and contexts are used to specify those semantic properties that have to be preserved by migration processes. In Sect. 2.2 of our introductory survey we have used the concept EntryPoint in order to highlight some important design goals. Concepts *group different implementations* of a property into different contexts. Also, a concept's interface defines *role names* for its interface objects. Only lifecycles of these objects are traced when checking preservation requirements. All concepts and contexts have names that are globally unique. Hence, they can be referenced globally without referring to a detailed implementation directly. In this way, we *abstract* from concrete implementations and can use concepts and contexts as language elements in the high-level preservation language that will be introduced in Chap. 5. In the following sections we will provide the necessary formalisms. The agenda is as follows:

- As usual, we start with an informal survey in Sect. 4.1.
- In Sect. 4.2 we introduce formal concept specifications. They comprise context definitions, concept definitions, interface definitions, role names, and role-assignments (cf. Sect. 2.2 of the introductory survey of this thesis).
- Section Sect. 4.3 deals with *dynamic* concept evaluation. As opposed to static term values and formula semantics, term and formula evaluation is reduced to the set of existing objects in the dynamic setting. We will prove important correspondence properties between static and dynamic evaluation results. Finally, we use dynamic term values and dynamic formula semantics for the formal definition of concept terms, concept formulas, and their semantics.
- The results are summarized in Sect. 4.4.

4.1 Informal Overview

Interfacing mechanisms and formal interface description languages (*IDL*s, e.g., CORBA IDL or Java IDL) are well-known in Computer Science. In general, these languages describe interface functionality by typing constraints and pre- and post-conditions. They, however, usually do not "know" their concrete implementations. In digital archiving, however, the latter is normally required. As an example, most DAs support archiving formats that are open standards only. Keeping digital objects in proprietary formats is



concept EntryPoint =
begin
Interface:
$\langle$**website** : α_w, **sourcedir** : α_d,
homepage : $\alpha_h\rangle$
α_w<`Website` $\wedge$ α_d<`Dir` $\wedge$
α_h<`HTMLDoc`
Context *AWeb*:
ι_A | *validAWeb*(*website*) $\wedge$
sourcedir = `srcDir`(*website*) $\wedge$
homepage = `home`(*website*)
Context *BWeb*:
ι_B | *validBWeb*(*website*) $\wedge$
sourcedir = `srcDir`(*website*) $\wedge$
homepage = `home`(*website*)
end EntryPoint;

Figure 4.1: Specification of concept EntryPoint and website names

commonly considered to be too risky as it hinders (if not prevents) access to the internal structure of the hosted objects; the archive would become dependent on the (usually commercial) owners of these formats. Therefore, concepts in our setting comprise both, a well-defined interface and all known implementations.

In the left-hand part of Fig. 4.1 we have depicted two "visual concept specifications". Recall that we distinguish two variants of preservation. The concept EntryPoint (upper left-hand part) captures a *semantic relationship*. The lower left-hand part shows the Name concept. It is functional and covers *object content*. On the right-hand side we have listed the specification of EntryPoint in the pseudo language that will be used in this thesis (as we will cover both concept variants similarly, we omit the other specification for brevity). The specification consists of the following parts:

- The concept's *name* occurs at the top-most line. We will require this name to be globally unique later on.
- The *interface* is specified within the **Interface** part. It defines an *arity* by means of a sequence of (generically) typed role names. These role names are marked bold and are distinguished from regular variables that are used in *concept implementations*. An additional formula is used to impose further type constraints. In our example, we permit the concept EntryPoint to hold between a website (type `Website`), a directory, and an html document only.
- The specification introduces two *contexts* *AWeb* and *BWeb*. These names have to be globally unique as well. The *implementations* ι_A and ι_B are regular FOPL formulas.

Later on, the source and target objects of migration processes are matched w.r.t. concept interfaces in order to check desired preservation requirements. Typing can be seen as a pre-filter and speeds up the matching process. If the type check fails, matching fails immediately. If the type check succeeds, the implementing formulas (the contexts) are evaluated. The predicates *validAWeb* and *validBWeb* in Fig. 4.1 return `True` if a website

corresponds to the *AWeb* or *BWeb* format, respectively (cf. preservation requirements in the introductory survey to this thesis). The attributes `srcDir` and `home` return the source directory and home page, respectively, of a website. In the left-hand part of Fig. 4.1 we have shaded those objects that are affected by these format requirements. An object tuple *satisfies* a concept, if one of the concept's implementations evaluates to `True`.

Notice that we deliberately keep the specification scheme for concepts simple. We could, e.g., have introduced more genericness like is done in most general-purpose ADT specification languages like CASL. They allow for generic predicates and functions as well. Yet we have had no obligation to introduce these mechanisms up to now. But, it is clearly a point of future extensions.

Since all datatypes are specified *statically*, the question arises of how to evaluate these static formulas in a dynamic environment where objects may not exist; we would expect that only *existing objects* are considered. In our system we evaluate concept implementations *dynamically* modulo the set `existDObj`. The value of a non-existing object is "undefined". Functions applied to a non-existing object produce undefined values as well. Moreover, all predicates fail when applied to non-existing objects.

Example 4.1.1 (Dynamic term and formula evaluation)

In Tab. 4.1 we have listed the results of evaluating some example terms and formulas in the different system states (represented by the set of existing objects) of the example archive evolution in Tab. 3.1 (page 30). In order to abbreviate notation, we have named the respective terms as follows:

$$
\begin{array}{lcl}
\mathit{starthtml} & = & \texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, ...) \\
\mathit{calcdir} & = & \texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\}) \\
\mathit{overviewdir} & = & \texttt{Dir}(\texttt{nextID}^2(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{\}) \\
\mathit{sourcedir} & = & \texttt{Dir}(\texttt{nextID}^3(\texttt{initID}), \text{“}\texttt{source}\text{“}, \{\mathit{calcdir}, \mathit{overviewdir}\}, \{\mathit{starthtml}\})
\end{array}
$$

The terms t_1, t_2 and formulas ϕ_1, ϕ_2 are defined as follows:

$$
\begin{array}{lcl}
t_1 & := & \texttt{subDirs}(\mathit{sourcedir}) \\
t_2 & := & \mathit{calcdir} \\
\phi_1 & := & \mathit{overviewdir} = \mathit{overviewdir} \\
\phi_2 & := & \forall x : \texttt{Dir} \bullet \texttt{name}(x) = \text{“}\texttt{calc05}\text{“}
\end{array}
$$

Term t_1 returns all sub-directories of the "source" directory, t_2 corresponds to the "calc05" directory, ϕ_1 simply checks whether the directory "overview" equals itself, ϕ_2 checks if all directories have name "calc05".

□

For each of the terms/formulas Tab. 4.1 lists the static and dynamic evaluation result. In state zero no objects exist. Therefore, the semantics of t_1 is undefined whereas it equals the set $\{\mathit{calcdir}, \mathit{overviewdir}\}$ when evaluated statically. Also, the semantics of *calcdir* is undefined. Formula ϕ_1 always holds (indicated by `True`) in the static setting since the "overview" directory equals itself. However, this formula does not hold (indicated by `False`) when evaluated dynamically. Recall that we implement an existential variant of equality — equality never holds if one of the terms is undefined. The formula ϕ_2 holds in the dynamic setting as the quantifier sphere reduces to the set of existing objects (which is empty in state zero). ϕ_2, however, never holds when evaluated statically as directories may well have names other than "calc05". 7In state one, all terms and formulas are

Table 4.1: *Examples for dynamic term and formula semantics*

S	Value `existDObj`		Value t_1	Value t_2	Value ϕ_1	Value ϕ_2
0	$\{\}$	`static`	$\{calcdir, overviewdir\}$	$calcdir$	`True`	`False`
		`dynamic`	$\bot$	$\bot$	`False`	`True`
1	$\{starthtml\}$	`static`	$\{calcdir, overviewdir\}$	$calcdir$	`True`	`False`
		`dynamic`	$\bot$	$\bot$	`False`	`True`
2	$\{ starthtml, calcdir \}$	`static`	$\{calcdir, overviewdir\}$	$calcdir$	`True`	`False`
		`dynamic`	$\bot$	$calcdir$	`False`	`True`
3	$\left\{ \begin{matrix} starthtml, calcdir, \\ overviewdir \end{matrix} \right\}$	`static`	$\{calcdir, overviewdir\}$	$calcdir$	`True`	`False`
		`dynamic`	$\bot$	$calcdir$	`True`	`False`
4	$\left\{ \begin{matrix} starthtml, calcdir, \\ overviewdir, sourcedir \end{matrix} \right\}$	`static`	$\{calcdir, overviewdir\}$	$calcdir$	`True`	`False`
		`dynamic`	$\{calcdir, overviewdir\}$	$calcdir$	`True`	`False`

evaluated to the same results as in state zero as creating "start.html" does not affect them. In state two, the term *calcdir* becomes defined in the dynamic setting because the directory "calc05" is created. In state three ϕ_1 ($overview = overview$) holds since the directory "overview" has been created. Also, the quantifier sphere for ϕ_2 extends to $\{calcdir, overviewdir\}$ such that ϕ_2 does not hold anymore. Finally, t_1 becomes defined due to creation of the "source" directory. This example exhibits some properties of the dynamic term and formula semantics:

- Whenever being different from $\bot$, dynamic term values equal the corresponding static term values.
- Dynamic evaluation results for quantifier-free formulas equal the static results if all contained objects exist in the respective system state.
- Universally quantified formulas may hold in the dynamic setting even if they do not hold in the static setting.
- Quantified formulas may switch their truth values as the system evolves.

4.2 Specifying Contexts and Concepts

Here we introduce the *syntax* of concepts and their implementations in contexts. The next section then deals with concept *semantics*. To start with, we define subtype constraints over a signature Σ and type variables TV which we will need when defining concept interfaces.

Definition 4.2.1 (Type constraints) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$. Then a set TV of type variables is *suitable for* Σ, iff $TV \cap \mathcal{T} = \emptyset$. A *type binding* θ maps TV to $TV \cup \mathcal{T}$. The pointwise update is defined similar to variable assignments η.
The *simultaneous update* $\theta[\{\alpha_1 \mapsto \tau_1, ..., \alpha_n \mapsto \tau_n\}]$ *of* $\alpha_1, ..., \alpha_n \in TV \cup \mathcal{T}$ *in* θ *with* $\tau_1, ..., \tau_n \in \mathcal{T}$ is defined, iff $\forall i \neq j \in \{1, ..., n\} \bullet \alpha_i, \alpha_j \in TV \wedge \alpha_i = \alpha_j \Rightarrow \tau_i = \tau_j$. In this case,

$$\theta[\{\alpha_1 \mapsto \tau_1, ..., \alpha_n \mapsto \tau_n\}](\alpha) := \begin{cases} \tau_i, & \alpha = \alpha_i, i \in \{1, ..., n\} \\ \theta(\alpha), & \text{otherwise} \end{cases}$$

for $\alpha \in TV$.
Given a type binding θ and an element $\alpha \in TV \cup \mathcal{T}$. Then the *value* $\mathcal{V}^{\Sigma}[\![\alpha]\!]\theta$ *of* α *under type binding* θ is defined by

Table 4.2: *Syntax and semantics of type constraints over TV*

Syntax:	
(1) **Equality:** $\frac{s \in \mathcal{T} \cup TV \;\; t \in \mathcal{T} \cup TV}{s = t \in TC(TV, \Sigma)}$	$\mathcal{FV}(s = t) = \mathcal{FV}(s) \cup \mathcal{FV}(t)$
(2) **Subtyping:** $\frac{s \in \mathcal{T} \cup TV \;\; t \in \mathcal{T} \cup TV}{s<t \in TC(TV, \Sigma)}$	$\mathcal{FV}(s<t) = \mathcal{FV}(s) \cup \mathcal{FV}(t)$
(3) **Negation:** $\frac{\phi \in TC(TV, \Sigma)}{\neg\phi \in TC(TV, \Sigma)}$	$\mathcal{FV}(\neg\phi) = \mathcal{FV}(\phi)$
(4) **Conjunction:** $\frac{\phi \in TC(TV, \Sigma) \;\; \psi \in TC(TV, \Sigma)}{\phi \wedge \psi \in TC(TV, \Sigma)}$	$\mathcal{FV}(\phi \wedge \psi) = \mathcal{FV}(\phi) \cup \mathcal{FV}(\psi)$
Semantics:	
(1) **Equality:** $\frac{s = t \in TC(\emptyset, \Sigma) \;\; \mathcal{V}^{\Sigma}[\![s]\!]\theta \equiv \mathcal{V}^{\Sigma}[\![s]\!]\theta}{\Sigma \models s = t[\theta]}$	
(2) **Subtyping:** $\frac{s<t \in TC(\emptyset, \Sigma) \;\; (\mathcal{V}^{\Sigma}[\![s]\!]\theta, \mathcal{V}^{\Sigma}[\![t]\!]\theta) \in <}{\Sigma \models s<t[\theta]}$	
(3) **Negation:** $\frac{\neg\phi \in TC(\emptyset, \Sigma) \;\; not \; \Sigma \models \phi[\theta]}{\Sigma \models \neg\phi[\theta]}$	
(4) **Conjunction:** $\frac{\phi \wedge \psi \in TC(\emptyset, \Sigma) \;\; \Sigma \models \phi[\theta] \text{ and } \Sigma \models \psi[\theta]}{\Sigma \models \phi \wedge \psi[\theta]}$	

$$\mathcal{V}^{\Sigma}[\![\alpha]\!]\theta := \begin{cases} \theta(\alpha), & \alpha \in TV \\ \alpha, & \alpha \in \mathcal{T} \end{cases}$$

Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$ a suitable set TV of type variables, and a type binding θ. Then the set $TC(TV, \Sigma)$ of *type constraints over* Σ *and* TV, the set $\mathcal{FV}$ of *free variables*, and the semantics of type constraints are defined as shown in Tab. 4.2. □

Notice that we define a simultaneous update of type bindings. This will shorten notation in the following definitions. A simultaneous update is defined only, if the single updates do not clash. If there is $\alpha_i = \alpha_j$ and $\tau_i \neq \tau_j$, the simultaneous update is undefined. Now we are ready to introduce concept definitions.

Definition 4.2.2 (Contexts, interfaces, concepts) Given a specification $Spec := (\Sigma, Sen)$, $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$, such that $Spec$ is a permissible extension of the basic DA. Given furthermore a set TV of type variables suitable for Σ, a set of variables X suitable for Σ, and a set R of role names suitable for TV, Σ ($\{\alpha \mid \mathbf{r} : \alpha \in R\} \subseteq TV \cup \mathcal{T}$, $\{\mathbf{r} \mid \mathbf{r} : \alpha \in R\} \cap |\Sigma| = \emptyset$). Then the sets

(1) $CD^f(X, \Sigma)$ and $CD^{nf}(X, \Sigma)$ of *functional/non-functional context definitions over* X, Σ,

(2) $ID^f(R, TV, \Sigma)$ and $ID^{nf}(R, TV, \Sigma)$ of *functional/non-functional interface definitions over* R, TV, Σ, and

(3) $KD^f(R, TV, \Sigma)$ and $KD^{nf}(R, TV, \Sigma)$ of *functional/non-functional concept definitions over* R, TV, Σ

are defined as shown in Tab. 4.3.
Given an interface definition $\mathcal{I} \in ID(R, TV, \Sigma)$ that exactly contains roles $\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n$ in its signature. Then $role(i, \mathcal{I}) := r_i : \alpha_i$ determines the *role at position* i of $\mathcal{I}$. The set of all roles of $\mathcal{I}$ is determined by $roles(\mathcal{I}) := \{\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n\}$.
Given a context definition $C = \iota_C$ and an interface definition $\mathcal{I}$ with roles $\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} :$

Table 4.3: *Context, interface, and concept definitions over* TV, R, X, Σ

Syntax:

(1) **Context definitions:**

$$\frac{\begin{array}{l} C \notin N_X \cup |\Sigma| \\ \iota_C \in T_{\texttt{Top}}(X, \Sigma) \\ \tau\texttt{<DObj} \text{ for all } x : \tau \in \mathcal{FV}(\iota_C) \end{array}}{C = \iota_C \in CD^f(X, \Sigma)} \qquad \frac{\begin{array}{l} C \notin N_X \cup |\Sigma| \\ \iota_C \in F(X, \Sigma) \\ \tau\texttt{<DObj} \text{ for all } x : \tau \in \mathcal{FV}(\iota_C) \end{array}}{C = \iota_C \in CD^{nf}(X, \Sigma)}$$

(2) **Interface definitions:**

$$\frac{\begin{array}{l} \mathbf{r_1} : \alpha_1 \in R, ..., \mathbf{r_n} : \alpha_n \in R, \\ \forall i \neq j \in \{1, ..., n\} \bullet \mathbf{r_i} \neq \mathbf{r_j} \\ \alpha \in \mathcal{T} \cup TV, \phi \in TC(TV, \Sigma) \end{array}}{\begin{array}{l} (\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n) \to \alpha[\phi] \\ \in ID^f(R, TV, \Sigma) \end{array}} \qquad \frac{\begin{array}{l} \mathbf{r_1} : \alpha_1 \in R, ..., \mathbf{r_n} : \alpha_n \in R, \\ \forall i \neq j \in \{1, ..., n\} \bullet \mathbf{r_i} \neq \mathbf{r_j} \\ \phi \in TC(TV, \Sigma) \end{array}}{\begin{array}{l} (\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n)[\phi] \\ \in ID^{nf}(R, TV, \Sigma) \end{array}}$$

(3) **Concept definitions:**

$$\frac{\begin{array}{l} \mathcal{K} \notin N_R \cup N_X \cup |\Sigma| \cup \bigcup_i C_i, \\ X_1, ..., X_n \text{ suitable for } \Sigma, \\ C_1 = \iota_{C_1} \in CD^f(X_1, \Sigma), ..., C_n = \iota_{C_n} \in CD^f(X_n, \Sigma), \\ (\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n) \to \alpha[\phi] \in ID^f(R, TV, \Sigma), \\ C_1 = \iota_{C_1} \text{ implements } (\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n) \to \alpha[\phi], ..., \\ C_n = \iota_{C_n} \text{ implements } (\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n) \to \alpha[\phi] \end{array}}{\mathcal{K}(\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n) \to \alpha[\phi] = \{C_1 = \iota_{C_1}, ..., C_n = \iota_{C_n}\} \in KD^f(R, TV, \Sigma)}$$

$$\frac{\begin{array}{l} \mathcal{K} \notin N_R \cup N_X \cup |\Sigma| \cup \bigcup_i C_i, \\ X_1, ..., X_n \text{ suitable for } \Sigma, \\ C_1 = \iota_{C_1} \in CD^{nf}(X_1, \Sigma), ..., C_n = \iota_{C_n} \in CD^{nf}(X_n, \Sigma), \\ (\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n)[\phi] \in ID^{nf}(R, TV, \Sigma), \\ C_1 = \iota_{C_1} \text{ implements } (\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n)[\phi], ..., \\ C_n = \iota_{C_n} \text{ implements } (\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n)[\phi] \end{array}}{\mathcal{K}(\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n)[\phi] = \{C_1 = \iota_{C_1}, ..., C_n = \iota_{C_n}\} \in KD^{nf}(R, TV, \Sigma)}$$

α_n, the *role assignment* $ra(C, \mathcal{I}) : \mathcal{FV}(\iota_C) \to roles(\mathcal{I})$ *for* C *and* $\mathcal{I}$ is defined by

$$ra(C, \mathcal{I})(x : \tau) := \begin{cases} \mathbf{r} : \alpha, & \mathbf{r} : \alpha \in roles(\mathcal{I}), x = \mathbf{r} \\ undefined, & \text{otherwise} \end{cases} .$$

Given a type binding θ. A non-functional context $C = \iota_C$ over X and Σ *implements* a non-functional interface $\mathcal{I} := (\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n)[\phi]$ over Σ, TV iff

(1) $ra(C, \mathcal{I})$ is total and

(2) $\theta' := \theta[\{\alpha' \mapsto \tau' \mid x : \tau' \in \mathcal{FV}(\iota_C), ra(C, \mathcal{I})(x : \tau') = \mathbf{r} : \alpha')\}]$ is defined and $\phi[\theta']$ is satisfiable.

Given a type binding θ. A functional context $C = \iota_C$ over X and Σ, $\iota_C \in T_\tau(X, \Sigma)$, *implements* a functional interface $\mathcal{I} := (\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n) \to \alpha[\phi]$ over Σ, TV iff

(1) $ra(C, \mathcal{I})$ is total and

(2) $\theta' := \theta[\{\alpha' \mapsto \tau' \mid x : \tau' \in \mathcal{FV}(\iota_C), ra(C, \mathcal{I})(x : \tau') = \mathbf{r} : \alpha')\} \cup \{\alpha \mapsto \tau\}]$ is defined and $\phi[\theta']$ is satisfiable.

□

Whenever it plays no role whether we refer to functional or non-functional context, interface, and concept definitions, we refer to them by $CD(X, \Sigma)$, $ID(R, TV, \Sigma)$,

and $KD(R, TV, \Sigma)$, respectively. Concept definitions have similarities to function declarations in programming languages. Concept interfaces correspond to signatures and context definitions introduce implementations or function bodies. Context definitions comprise a context name that is globally unique and either a term over Σ,X or a formula over Σ,X. In the former case we speak of *functional contexts*, where the other variant is *non-functional*. The given term/formula of a context is the implementation and *embeds* this context into a concept. For this purpose, variables may occur free in contexts, but have to be of type `DObj` or below. Concept interfaces fix the (generic) signature of a concept and define role names. Again, we distinguish functional and non-functional interfaces. The set of permissible instantiations can be restricted by a type constraint ϕ.

All contexts of a concept have to *implement* the concept's interface. Non-functional interfaces may be implemented by non-functional contexts only. The same is true for functional interfaces and functional contexts. In both cases the definition above requires a name-preserving mapping from the free variables of the context's implementation to the roles of the interface. This mapping is called *role assignment* in analogy to the example survey in Sect. 2.2. It, however, is also sometimes referred to as fitting morphisms [CoF04] or signature morphisms [EGL89] in the literature. Since we do not map full signatures, but map variables to roles only, we have decided to use this more precise notion. Also, fitting morphisms are, in general, *not* name-preserving, which results in *sets* of suitable morphisms. In contrast, role assignments are uniquely determined. This speeds up concept matching. Also, notice that role assignments need not be surjective. Implementations can include less free variables than interfaces offer. We, however, require that all role-assignments respect the imposed type constraint ϕ. For this purpose, a type-binding θ is computed according to the role assignment. The type constraint must still be *satisfiable* under this type binding. This property has to be checked only once when contexts and concepts are specified. Performance is, thus, a minor issue. In the left-hand part of Fig. 4.1 on page 55 we have indicated role assignments by dotted arrows.

Concept definitions consist of an interface and context definitions. Depending on the type of interface (functional / non-functional) we distinguish functional and non-functional concepts. In either case, concept names have to be distinct from all context names and all symbol names of Σ. Also, all included context definitions must implement the concept's interface. Sample concept definitions for EntryPoint and Name can be found in App. B.2.

Notice that we permit concept implementations to contain *dynamic* functions (like `existDObj`). Concept satisfaction, thus, can depend on system states, which provides an additional and necessary degree of expressiveness and flexibility. We will provide an example in our case study (Part IV).

4.3 Evaluating Concepts

Since concept implementations are Σ-terms or Σ-formulas, concept semantics is strongly related to evaluating terms and formulas. However, concepts are evaluated in our *dynamic* environment where objects may not exist. In particular, non-existing objects are

Table 4.4: *Dynamic term semantics*

Dynamic term value:

(1) **Variables:** $$\frac{x : \tau \in X_\tau \;\; \eta \in Env(X, \mathcal{A})}{(\mathcal{A}, \eta, x) \overset{t_d}{\rightsquigarrow} \eta(x)}$$

(2) **Function application:**

(2.1) $$\frac{\begin{array}{ll} f(t_1, ..., t_n) \in T_\tau(X, \Sigma) & \mathcal{A} \in A(\Sigma), \eta \in Env(X, \mathcal{A}) \\ \tau{<}\texttt{DObj} & \mathcal{A} \models \neg(f(t_1, ..., t_n) \in \texttt{existDObj})[\eta] \end{array}}{(\mathcal{A}, \eta, f(t_1, ..., t_n)) \overset{t_d}{\rightsquigarrow} \bot}$$

(2.2) $$\frac{\begin{array}{ll} f(t_1, ..., t_n) \in T_\tau(X, \Sigma) & \mathcal{A} \in A(\Sigma), \eta \in Env(X, \mathcal{A}) \\ \tau{<}\texttt{DObj} & \mathcal{A} \models f(t_1, ..., t_n) \in \texttt{existDObj}[\eta] \\ & (\mathcal{A}, \eta, f(t_1, ..., t_n)) \overset{t}{\rightsquigarrow} v \end{array}}{(\mathcal{A}, \eta, f(t_1, ..., t_n)) \overset{t_d}{\rightsquigarrow} v}$$

(2.3) $$\frac{\begin{array}{ll} f(t_1, ..., t_n) \in T_\tau(X, \Sigma) & \mathcal{A} \in A(\Sigma), \eta \in Env(X, \mathcal{A}) \\ \neg\tau{<}\texttt{DObj} & (\mathcal{A}, \eta, f(t_1, ..., t_n)) \overset{t}{\rightsquigarrow} v \\ c(t'_1, ..., t'_m) \in CT^{min}_\tau(\Sigma, v) & (\mathcal{A}, \eta, t'_1) \overset{t_d}{\rightsquigarrow} v_1, ..., (\mathcal{A}, \eta, t'_m) \overset{t_d}{\rightsquigarrow} v_m \end{array}}{(\mathcal{A}, \eta, f(t_1, ..., t_n)) \overset{t_d}{\rightsquigarrow} c^{\mathcal{A}^\perp}(v_1, ..., v_m)}$$

expected to have "undefined" semantics.

In the following we introduce a dynamic semantics for Σ-terms and Σ-formulas. It will both, adhere to the requirements explained in the introduction to this chapter, and cause a "minimal" influence to static term and formula evaluation only. In particular, we can prove strong correspondence theorems to the static semantics. Finally, we will introduce formal concept terms and concept formulas together with their semantics.

Dynamic Term Values

The term structure of Σ-terms with *dynamic term value* is defined as follows.

Definition 4.3.1 (Dynamic values) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA and a set X of variables suitable for Σ. Then the structure $T^d_{X,\Sigma}$ of *terms over* Σ *with dynamic value* is defined by

$$T^d_{X,\Sigma} := (T_{\texttt{Top}}(X, \Sigma), \overset{t_d}{\rightsquigarrow}, \mathcal{FV})$$

where the components $T_{\texttt{Top}}(X, \Sigma)$ and $\mathcal{FV}$ are determined as shown in Tab. 3.2, page 37, and $\overset{t_d}{\rightsquigarrow}$ is defined as shown in Tab. 4.4. In analogy to static term values, we write $v = \mathcal{V}^{\mathcal{A}}_d[\![t]\!]\eta$ for $(\mathcal{A}, \eta, t) \overset{t_d}{\rightsquigarrow} v$. □

First, notice that η may contain an update $x \mapsto a$ where a is in the type domain $\texttt{DObj}^{\mathcal{A}}$. When evaluating $\mathcal{V}^{\mathcal{A}}_d[\![t]\!]\eta$ we, thus, have to consider the case that a is the semantics of a term t that does not exist in the system (i.e., $t \notin \texttt{existDObj}$). In the sequel we will, however, always compute environments using the dynamic semantics; if $a \neq \bot$, t exists in the system. Therefore, we do not explicitly distinguish this case and keep our semantics simple.

Function application is separated into three rules:

(2.1) If $f(t_1, ..., t_n)$ is of type `DObj` and the result is *not* in `existDObj` (w.r.t. the *static* semantics), this term yields $\bot$. This directly realizes our intention not to permit non-existing objects. As an example, confer row one of the example in Tab. 4.1 on page 57. No objects exist and, hence, *calcdir* evaluates to $\bot$. Since all functions are strict in our setting, superior function calls also yield $\bot$ if applied to this result.

(2.2) If $f(t_1, ..., t_n)$ is of type `DObj` and the result is in `existDObj`, the *static* value is returned. Again, this does not assure our intended semantics directly. Generally, one of the sub-terms might contain non-existing digital objects. We, however, evaluate concepts only in our dynamic environment. There, state changes assure the system invariants stated in Lemma 3.3.5 on page 51. In particular, all object-valued content of all existing objects also exists.

(2.3) If $f(t_1, ..., t_n)$ is of type `DObj`, we generate a constructor term of minimal length that represents $f(t_1, ..., t_n)$. Recall that these constructor terms contain all relevant object-valued content for $f(t_1, ..., t_n)$ as the constructor property holds in all permissible extensions of the basic DA. When evaluating this constructor term we, thus, automatically assure that all relevant sub-objects of the original term $f(t_1, ..., t_n)$ are checked for existence.

In the following, we prove correspondence between $\overset{t}{\rightsquigarrow}$ and $\overset{t_d}{\rightsquigarrow}$. We start by showing that the static term values equal dynamic term values if the latter are not $\bot$.

Lemma 4.3.1 Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA, a set X of variables that is suitable for Σ, a type $\tau \in \mathcal{T}$, a term $t \in T_\tau(X, \Sigma)$, a Σ-model $\mathcal{A}$, and a variable assignment η. If the dynamic term value of t in $\mathcal{A}$ under η defined, it equals the static term value of t, i.e..

$$\mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta \neq \bot \Rightarrow \mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta.$$

□

The proof can be found in App. C.2, page 210, and goes by straightforward induction on the structure of t. With this prerequisite, we can prove the following strong correspondence between static and dynamic term values.

Theorem 4.3.1 (Dynamic term values correspond to static term values) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA and a set X of variables suitable for Σ. Given furthermore a Σ-model $\mathcal{A}$ that is derived by basic transitions, a variable assignment η such that for all $x \mapsto a \in \eta$, $a \in \tau^{\mathcal{A}}$, it is true that $\mathcal{V}_d^{\mathcal{A}}[\![t']\!] = a$ for $t' \in CT_{\mathcal{A}}^{min}(\Sigma, a)$, and a term $t \in T_\tau(X, \Sigma)$ such that $\mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta \neq \bot$. Then the following holds:

(1) Static and dynamic term value of a term are equal iff all objects in the object-valued content of that term exist, i.e.,

$$\mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta \Leftrightarrow \mathcal{A} \models \forall y : \texttt{DObj} \bullet y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(t) \Rightarrow y \in \texttt{existDObj}[\eta],$$

(2) The dynamic term value for t is undefined iff not all objects in the object-valued content of that term exist, i.e.,

$$\mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = \bot \Leftrightarrow \mathcal{A} \models \exists y : \texttt{DObj} \bullet y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(t) \ \wedge \ y \notin \texttt{existDObj}[\eta],$$

There $y \notin \mathcal{FV}(t)$.

□

This theorem applies to *functional* concepts; their semantics will be determined using dynamic term values later on. The proof in App. C.2, page 211, goes by induction on the structure of t and exploits the fact that $\mathcal{A}$ is derived by basic transitions. This property holding implies that the system invariants of Lemma 3.3.5 hold. Using part (1) of the theorem above and Lemma 4.3.1, we can also prove part (2). It states that the dynamic term semantics shows the intended behavior in presence of non-existing objects in the object-valued content. The formal arguments are also provided in App. C.2.

The strong correspondence between static and dynamic term values together with Lemma 3.2.2 ($T_{X,\Sigma}$ is well-defined) imply that $T^d_{X,\Sigma}$ is well-defined. This is fixed in the following corollary.

Corollary 4.3.1 (Dynamic term values are well-defined) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$ and a set X of variables suitable for Σ. Then $T^d_{X,\Sigma}$ is a well-defined term structure and $\overset{t_d}{\rightsquigarrow}$ is a function, i.e.,

(1) For all $\tau, \tau' \in \mathcal{T}$ it is true that $\tau{<}\tau'$ implies $T_\tau(X,\Sigma) \subseteq T_{\tau'}(X,\Sigma)$.

(2) For all types $\tau \in \mathcal{T}$ and all terms in $T_\tau(X,\Sigma)$, $\mathcal{A} \in A(\Sigma)$, and $\eta \in Env(X,\mathcal{A})$ there is a unique $v \in \tau^{\mathcal{A}} \cup \{\bot\}$ such that $(\mathcal{A},\eta,t) \overset{t}{\rightsquigarrow} v$.

□

In App. B.2, page 185, we refine the introductory explanations on dynamic term values.

Dynamic Formula Semantics

The next definition introduces the *dynamic semantics* of formulas. It strongly bases on dynamic term values.

Definition 4.3.2 (Dynamic formula semantics) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA and a set X of variables suitable for Σ. Then the formula structure $F^d_{X,\Sigma}$ of *formulas over* Σ *with dynamic semantics* is defined by

$$F^d_{X,\Sigma} := FOL(AT(T^d_{X,\Sigma}), dom).$$

There,

$$dom(\mathcal{A}, x:\tau, \eta) := \begin{cases} (\mathcal{A}, \{v \mid \ v \in \tau^{\mathcal{A}}, ct \in CT^{min}_{\mathcal{A}}(\Sigma, v), \\ \qquad\qquad (\mathcal{A},\eta,ct) \overset{t_d}{\rightsquigarrow} v\}) & , \ \eta \in Env(X,\mathcal{A}) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

for all $(\mathcal{A}, x:\tau, \eta) \in A(\Sigma) \times X \times Env(X,\Sigma)$.
We identify the components of $F^d_{X,\Sigma}$ by $F(X,\Sigma)$ (set of formulas), $\models_d$ (satisfaction relation), and $\mathcal{FV}$ (free variables), respectively. $\mathcal{A} \not\models_d \phi[\eta]$ denotes the fact that $\mathcal{A} \models_d \phi[\eta]$ cannot be derived. □

The definition largely corresponds to the static case. It, however, uses the dynamic term structure $T^d_{X,\Sigma}$ and, thus, $\overset{t_d}{\rightsquigarrow}$ instead of $\overset{t}{\rightsquigarrow}$. Also quantifier spheres are evaluated differently. We quantify over all values that are typed appropriately *and* exist in the system. For this purpose, we take constructor terms of minimal length and derive their dynamic value. If this value is defined (and, hence, equals v according to Lemma 4.3.1), it is a member of the quantifier sphere. In App. B.2 we exemplify how *dom* works.

Analogous to the dynamic term values, our dynamic formula semantics is well-defined. This is stated formally in the next lemma. There, we also fix the observation that quantifier spheres for object types τ<`DObj` reduce to $\texttt{existDObj}^{\mathcal{A}} \cap \tau^{\mathcal{A}}$. This directly reflects our intention to evaluate formulas modulo the set of existing objects.

Lemma 4.3.2 (Dynamic formula semantics well-defined) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$ and a set X of variables suitable for Σ. Then (1) $F^d_{X,\Sigma}$ is a well-defined formula structure and (2) the quantifier spheres for object types reduce to the set of existing objects, i.e.,

(1) For all $\mathcal{A} \in A(\Sigma)$, $x : \tau \in X$, and $\eta \in Env(X, \Sigma)$ it is true that $dom(\mathcal{A}, x : \tau, \eta) = (\mathcal{A}, d)$ implies $\eta \in Env(X, \mathcal{A})$ and $d \subseteq \tau^{\mathcal{A}}$.

(2) For all $\mathcal{A} \in A(\Sigma)$ such that $\mathcal{A}$ is derived by basic transitions, $x : \tau \in X$, τ<`DObj`, and $\eta \in Env(X, \Sigma)$ it is true that $dom(\mathcal{A}, x : \tau, \eta) = (\mathcal{A}, d)$ implies

$$d = \tau^{\mathcal{A}} \cap \{\mathcal{V}^{\mathcal{A}}_d[\![t]\!] \mid t \in GT_\tau(\Sigma), \mathcal{A} \models t \in \texttt{existDObj}\}.$$

□

The proof can be found in App. C.2, page 212. Again, we provide a correspondence theorem that relates static and dynamic formula semantics.

Theorem 4.3.2 (Static and dynamic formula semantics correspond) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA and a set X of variables suitable for Σ. Given furthermore a Σ-model $\mathcal{A}$ that is derived by basic transitions, a variable assignment η, and a formula $\phi \in F(X, \Sigma)$ such that ϕ has no quantifiers and all terms occurring in ϕ do not evaluate to $\bot$. We define the *set T_ϕ of terms in ϕ* as follows:

$$\begin{array}{ll} \phi \equiv s = t : & T_\phi := \{s, t\} \\ \phi \equiv p(t_1, ..., t_n) : & T_\phi := \{t_1, ..., t_n\} \\ \phi \equiv \neg\psi : & T_\phi := T_\psi \\ \phi \equiv \psi \wedge \psi' : & T_\phi := T_\psi \cup T_{\psi'} \end{array}$$

Then the following holds:

(1) The dynamic semantics of ϕ equals the static semantics, if all objects in the object-valued content of the terms of ϕ exists in $\mathcal{A}$, i.e.,

$$\mathcal{A} \models \forall y : \texttt{DObj} \bullet y \in \bigcup_{t \in T_\phi} cont^{\mathcal{A},\eta}_{\texttt{DObj}}(t) \Rightarrow y \in \texttt{existDObj}[\eta]$$
$$\Rightarrow$$
$$(\mathcal{A} \models \phi[\eta] \Leftrightarrow \mathcal{A} \models_d \phi[\eta])$$

(2) Given $x : \tau \in X$ such that $x \in \mathcal{FV}(\phi)$. Then $\forall x : \tau \bullet \phi$ holding in the static case implies that $\forall x : \tau \bullet \phi$ holds in the dynamic case, if all objects in the object-valued content of all terms of ϕ exist in $\mathcal{A}$, i.e.,

$$\mathcal{A} \models \forall y : \texttt{DObj} \bullet y \in \bigcup_{t \in T_\phi} cont^{\mathcal{A},\eta}_{\texttt{DObj}}(t) \Rightarrow y \in \texttt{existDObj}[\eta]$$
$$\Rightarrow$$
$$(\mathcal{A} \models \forall x : \tau \bullet \phi[\eta] \Rightarrow \mathcal{A} \models_d \forall x : \tau \bullet \phi[\eta])$$

(3) Given $x : \tau \in X$ such that $x \in \mathcal{FV}(\phi)$ and $\tau <: \texttt{DObj}$. Then $\mathcal{A} \models \forall x : \tau \bullet x \in \texttt{existDObj} \Rightarrow \phi[\eta]$ iff $\mathcal{A} \models_d \forall x : \tau \bullet \phi[\eta]$ holds if all objects in the object-valued content of the terms of ϕ exist in $\mathcal{A}$, i.e.,

$$\mathcal{A} \models \forall y : \texttt{DObj} \bullet y \in \bigcup_{t \in T_\phi} cont^{\mathcal{A},\eta}_{\texttt{DObj}}(t) \Rightarrow y \in \texttt{existDObj}[\eta]$$
$$\Rightarrow$$
$$(\mathcal{A} \models_d \forall x : \tau \bullet \phi[\eta] \Leftrightarrow \mathcal{A} \models \forall x : \tau \bullet x \in \texttt{existDObj} \Rightarrow \phi[\eta])$$

□

The proof can be found in App. C.2, page 213. The results of this theorem together with Thm. 4.3.1 apply to *non-functional* concepts; their semantics will be defined using $\models_d$ later on.

As quantifier spheres may shrink when switching from the static to the dynamic setting, truth-values of universally quantified formulas generally do not carry over equivalently to the dynamic case. Yet valid, universally quantified formulas (w.r.t. their static semantics) are always valid in the dynamic case when all object-valued content of all terms in the body ϕ exists. This correspondence becomes especially clear with property (3). On the left-hand side we do not have to require $x \in \texttt{existDObj}$ because the dynamic quantifier sphere automatically reduces to the elements in `existDObj`. The truth value of this formula then is equivalent to the (static) truth value of the extended variant using $x \in \texttt{existDObj}$. Since formulas $\exists x \bullet \phi$ are equivalent to $\neg\forall x \bullet \neg\phi$, the dual results of Thm. 4.3.2(2) hold for existentially quantified formulas. In particular, those formulas $\exists x \bullet \phi$ that do *not* hold in the static setting do not hold in the dynamic setting as well. Due to the correspondence theorems we know which results carry over from the static to the dynamic environment and under which circumstances they do. As an advantage, we can employ automated theorem provers that might not work for evolving systems. In each system state we know the set of existing objects and can decide if these so-proved properties carry over. As an example, the system invariants listed in Cor. 3.3.1, page 52, hold in the dynamic setting as well due to Thm. 4.3.2.

Syntax and Semantics of Concept Expressions

Syntax and semantics of *concept expressions*, i.e., *concept terms* and *concept formulas*, is introduced next. Concept expressions occur in formal preservation requirements, which will be introduced in Chap. 5.

Definition 4.3.3 (Concept expressions) Given a specification $Spec := (\Sigma, Sen)$, $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$, such that $Spec$ is a permissible extension of the basic DA. Given furthermore a set TV of type variables suitable for Σ, a set of variables X suitable for Σ, a set R of role names suitable for TV, Σ, and a set $KD \subseteq KD(R, TV, \Sigma)$ of non-overloaded

Table 4.5: *Concept terms and concept formulas over* X, Σ, KD

Syntax:

(1) **Concept terms with context:**

$$\frac{\begin{array}{l} t_1 \in T_{\tau_1}(X,\Sigma), ..., t_n \in T_{\tau_n}(X,\Sigma) \\ \mathcal{K}\,\mathcal{I} = \{C_1 = \iota_{C_1}, ..., C_m = \iota_{C_m}\} \in KD \cap KD^f(R, TV, \Sigma) \\ \mathcal{I} \equiv (\mathbf{r_1} : \alpha_1, ..., \mathbf{r_n} : \alpha_n) \to \alpha[\phi] \\ C = \iota_C \in \{C_1 = \iota_{C_1}, ..., C_m = \iota_{C_m}\}, \iota_C \in T_\tau(X', \Sigma) \\ \Sigma \models \phi[\{\alpha_i \mapsto \tau_i \mid i \in \{1, ..., n\}, \mathbf{r_i} : \alpha_i = \mathit{role}(i, \mathcal{I})\} \cup \{\alpha \mapsto \tau\}] \\ \forall x : \tau' \in \mathcal{FV}(\iota_C), i \in \mathbb{N} \bullet \mathit{ra}(C, \mathcal{I})(x : \tau') = \mathit{role}(i, \mathcal{I}) \Rightarrow \tau_i{<}\tau' \end{array}}{\mathcal{K}(t_1, ..., t_n)[C] \in KT_\tau(X, \Sigma, KD)}$$

$$\mathcal{FV}(\mathcal{K}(t_1, ..., t_n)[C]) = \bigcup_{1 \le i \le n} \mathcal{FV}(t_i)$$

(2) **Concept terms with wildcard:**

$$\frac{\begin{array}{l} t_1 \in T_{\tau_1}(X,\Sigma), ..., t_n \in T_{\tau_n}(X,\Sigma) \\ \mathcal{K}\,\mathcal{I} = \{C_1 = \iota_{C_1}, ..., C_m = \iota_{C_m}\} \in KD \cap KD^f(R, TV, \Sigma) \\ \exists C \in \{C_1, ..., C_n\} \bullet \mathcal{K}(t_1, ..., t_n)[C] \in KT_\tau(X, \Sigma, KD) \end{array}}{\mathcal{K}(t_1, ..., t_n)[\_] \in KT_\tau(X, \Sigma, KD)}$$

$$\mathcal{FV}(\mathcal{K}(t_1, ..., t_n)) = \bigcup_{1 \le i \le n} \mathcal{FV}(t_i)$$

(2) **Subtyping:**

$$\frac{t \in KT_\tau(X, \Sigma, KD)\ \tau{<}\tau'}{t \in KT_{\tau'}(X, \Sigma, KD)}$$

Concept terms:

(1) **Concept formulas with context:**

$$\frac{\begin{array}{l} t_1 \in T_{\tau_1}(X,\Sigma), ..., t_n \in T_{\tau_n}(X,\Sigma) \\ \mathcal{K}\,\mathcal{I} = \{C_1 = \iota_{C_1}, ..., C_m = \iota_{C_m}\} \in KD \cap KD^{nf}(R, TV, \Sigma) \\ C = \iota_C \in \{C_1 = \iota_{C_1}, ..., C_m = \iota_{C_m}\} \\ \Sigma \models \phi[\{\alpha_i \mapsto \tau_i \mid i \in \{1, ..., n\}, \mathbf{r_i} : \alpha_i = \mathit{role}(i, \mathcal{I})\}] \\ \forall x : \tau' \in \mathcal{FV}(\iota_C), i \in \mathbb{N} \bullet \mathit{ra}(C, \mathcal{I})(x : \tau') = \mathit{role}(i, \mathcal{I}) \Rightarrow \tau_i{<}\tau' \end{array}}{\mathcal{K}(t_1, ..., t_n)[C] \in KF(X, \Sigma, KD)}$$

$$\mathcal{FV}(\mathcal{K}(t_1, ..., t_n)[C]) = \bigcup_{1 \le i \le n} \mathcal{FV}(t_i)$$

(2) **Concept formulas with wildcards:**

$$\frac{\begin{array}{l} t_1 \in T_{\tau_1}(X,\Sigma), ..., t_n \in T_{\tau_n}(X,\Sigma) \\ \mathcal{K}\,\mathcal{I} = \{C_1 = \iota_{C_1}, ..., C_m = \iota_{C_m}\} \in KD \cap KD^{nf}(R, TV, \Sigma) \\ \exists C \in \{C_1, ..., C_n\} \bullet \mathcal{K}(t_1, ..., t_n)[C] \in KF(X, \Sigma, KD) \end{array}}{\mathcal{K}(t_1, ..., t_n)[\_] \in KF(X, \Sigma, KD)}$$

$$\mathcal{FV}(\mathcal{K}(t_1, ..., t_n)) = \bigcup_{1 \le i \le n} \mathcal{FV}(t_i)$$

concept definitions. Then the set of *concept terms of* $KT_\tau(X, \Sigma, KD)$ *of type* τ *over* X, Σ, KD, the set of *concept formulas* $KF(X, \Sigma, KD)$ *over* X, Σ, KD, and the set $\mathcal{FV}$ of free variables of concept terms and concept formulas are defined as shown in Tab. 4.5. □

While concept terms can be used like regular terms, concept formulas are used like predicates in regular Σ-formulas. According to Tab. 4.5, concept expressions need not necessarily refer to an explicit context $[C]$, but can include a *wildcard* _. Concept expressions that contain wildcards are well-formed if they are well-formed for at least one of their contexts. This may seem to be syntactic sugar. Later on, however, we will use concept expressions in our preservation requirements. When a new implementation is added, it is automatically considered in the concept's semantics if wildcards are used (due to the built in universal quantification). Existing specifications, thus, need not be changed in that case.

Concept terms and concept formulas satisfy similar well-formedness and well-typedness

rules. The well-typedness rules for concept terms comprise two conditions. First,

$$\Sigma \models \phi[\{\alpha_i \mapsto \tau_i \mid i \in \{1,...,n\}, \mathbf{r_i} : \alpha_i = \mathit{role}(i,\mathcal{I})\} \cup \{\alpha \mapsto \tau\}]$$

requires type constraint ϕ to hold under the induced type binding. In particular, this simultaneous update binds the type τ_i of term t_i to α_i, if α_i is a type variable and it is the type of role $\mathbf{r_i} : \alpha_i$ at position i in the concept's interface. This is, in general, not fully assured by the second typing condition

$$\forall x : \tau' \in \mathcal{FV}(\iota_C) \ \bullet \ \mathbf{r_i} : \alpha_i = \mathit{ra}(C,\mathcal{I})(x : \tau') \Rightarrow \tau_i{<}\tau'.$$

It basically assures that $t_1, ..., t_n$ are substituted correctly in the implementation ι_C. Using the role assignment $\mathit{ra}(C,\mathcal{I})$, we determine the position of the role that matches $x : \tau'$. Assuming this index is i, term i must be substitutable for $x : \tau'$. This requires the type τ_i of t_i to be a subtype of τ'.

Notice that no implementation details occur in concept expressions. The concrete implementation may change without necessarily inducing a change to existing specifications. This is an important design goal of our approach.

In the following definition we introduce the semantics of concept terms and concept formulas. It strongly relies on dynamic term and formula semantics.

Definition 4.3.4 (Semantics of concept expressions) Given a specification $\mathit{Spec} := (\Sigma, \mathit{Sen})$, $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$, such that Spec is a permissible extension of the basic DA. Given furthermore a set TV of type variables suitable for Σ, a set of variables X suitable for Σ, a set R of role names suitable for TV, Σ, and a set $KD \subseteq KD(R, TV, \Sigma)$ of non-overloaded concept definitions. Then the semantics of concept terms and concept formulas over X, Σ, KD is defined as shown in Tab. 4.6.
Moreover, *the term structure* $KT_{X,\Sigma,KD}$ *of concept terms over* X, Σ, KD and the *formula structure* $KF_{X,\Sigma,KD}$ *of concept formulas over* X, Σ, KD are defined by

$$\begin{aligned} KT_{X,\Sigma,KD} &:= (KT_{\mathtt{Top}}(X,\Sigma,KD), \overset{t_d}{\rightsquigarrow}, \mathcal{FV}) \\ KF_{X,\Sigma,KD} &:= (KF(X,\Sigma,KD), A(\Sigma), \models_d, \mathcal{FV}) \end{aligned}$$

where the components are determined as shown in Tab. 4.5 and Tab. 4.6. □

First, notice that we overload $\overset{t_d}{\rightsquigarrow}$, $\models_d$, and $\mathcal{FV}$. Recall that we have required concept names to be distinct from all symbols in Σ. Therefore, the sets of concept terms and regular Σ-terms on the one hand, and concept formulas and Σ-formulas on the other hand, are disjoint. Hence, overloading the just-mentioned symbols does not lead to ambiguity. Re-using them, however, underlines that we employ the dynamic semantics when evaluating concept terms and concept formulas.

The concept value for $\mathcal{K}$ in C yields the *dynamic* value of the implementing term ι_C of C. Since all context-names are pairwise distinct, this context and it's implementing term are uniquely determined. In order to evaluate ι_C, we need to generate an appropriate variable update η for the free variables of ι_C. For this purpose, we use the role assignment $\mathit{ra}(C,\mathcal{I})$. First, we determine a mapping l from the free variables of ι_C to the indices of their matching roles in $\mathcal{I}$. If x_j belongs to role with index $l(x_j)$ of $\mathcal{I}$, we know that term $t_{l(x_j)}$ is to be used. The update for x_j then yields $x_j \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{(l(x_j))}]\!]\eta$. This is done

Table 4.6: *Semantics of concept terms and concept formulas over* X, Σ, KD

Concept terms:

(1) **Concept terms with context:**

$$\frac{\begin{array}{l}\mathcal{K}(t_1,...,t_n)[C] \in KT_\tau(X,\Sigma,KD)\\ \mathcal{K}\ \mathcal{I} = \{C_1 = \iota_{C_1},...,C_m = \iota_{C_m}\} \in KD\\ C = \iota_C \in \{C_1 = \iota_{C_1},...,C_m = \iota_{C_m}\}\\ \mathcal{FV}(\iota_C) = \{x_1,...,x_k\}\\ l : \mathcal{FV}(\iota_C) \to \{1,...,n\}\\ ra(C,\mathcal{I})(x_1) = role(l(x_1),\mathcal{I}),...,\\ ra(C,\mathcal{I})(x_k) = role(l(x_k),\mathcal{I})\end{array} \quad \begin{array}{l}(\mathcal{A},\eta,t_{l(x_1)}) \overset{t_d}{\rightsquigarrow} v_{l(x_1)},...,\\ (\mathcal{A},\eta,t_{l(x_k)}) \overset{t_d}{\rightsquigarrow} v_{l(x_k)}\\ (\mathcal{A},\eta[x_1 \mapsto v_{l(x_1)}]...[x_k \mapsto v_{l(x_k)}],\iota_C) \overset{t_d}{\rightsquigarrow} v\end{array}}{(\mathcal{A},\eta,\mathcal{K}(t_1,...,t_n)[C]) \overset{t_d}{\rightsquigarrow} v}$$

(2) **Concept terms with wildcard:**

$$\frac{\begin{array}{l}\mathcal{K}(t_1,...,t_n) \in KT_\tau(X,\Sigma,KD)\\ \mathcal{K}\ \mathcal{I} = \{C_1 = \iota_{C_1},...,C_m = \iota_{C_m}\} \in KD\end{array} \quad \begin{array}{l}\exists C \in \{C_1,...,C_m\} \bullet\\ \quad (\mathcal{A},\eta,\mathcal{K}(t_1,...,t_n)[C]) \overset{t_d}{\rightsquigarrow} v\end{array}}{(\mathcal{A},\eta,\mathcal{K}(t_1,...,t_n)[\_]) \overset{t_d}{\rightsquigarrow} v}$$

Concept formulas:

(1) **Concept formulas with context:**

$$\frac{\begin{array}{l}\mathcal{K}(t_1,...,t_n)[C] \in KF(X,\Sigma,KD)\\ \mathcal{K}\ \mathcal{I} = \{C_1 = \iota_{C_1},...,C_m = \iota_{C_m}\} \in KD\\ C = \iota_C \in \{C_1 = \iota_{C_1},...,C_m = \iota_{C_m}\}\\ \mathcal{FV}(\iota_C) = \{x_1,...,x_k\}\\ l : \mathcal{FV}(\iota_C) \to \{1,...,n\}\\ ra(C,\mathcal{I})(x_1) = role(l(x_1),\mathcal{I}),...,\\ ra(C,\mathcal{I})(x_k) = role(l(x_k),\mathcal{I})\end{array} \quad \begin{array}{l}(\mathcal{A},\eta,t_{l(x_1)}) \overset{t_d}{\rightsquigarrow} v_{l(x_1)},...,\\ (\mathcal{A},\eta,t_{l(x_k)}) \overset{t_d}{\rightsquigarrow} v_{l(x_k)}\\ \mathcal{A} \models_d \iota_C[\eta[x_1 \mapsto v_{l(x_1)}]...[x_k \mapsto v_{l(x_k)}]]\end{array}}{\mathcal{A} \models_d \mathcal{K}(t_1,...,t_n)[C][\eta]}$$

(2) **Concept formulas with wildcard:**

$$\frac{\begin{array}{l}\mathcal{K}(t_1,...,t_n) \in KF(X,\Sigma,KD)\\ \mathcal{K}\ \mathcal{I} = \{C_1 = \iota_{C_1},...,C_m = \iota_{C_m}\} \in KD\end{array} \quad \begin{array}{l}\exists C \in \{C_1,...,C_m\} \bullet\\ \quad \mathcal{A} \models_d \mathcal{K}(t_1,...,t_n)[C][\eta]\end{array}}{\mathcal{A} \models_d \mathcal{K}(t_1,...,t_n)[\_][\eta]}$$

for all free variables, which yields the overall variable assignment for the evaluation of ι_C. Notice that all this is well-defined only since $\mathcal{K}(t_1,...,t_n)[C]$ is well-formed and well-typed. In particular, $ra(C,\mathcal{I})$ *uniquely* assigns variables to role names in the concept interface. Also, both, role names and context names, are all pairwise distinct.

Due to the facility to use wildcards, concept terms may have multiple values, which may seem unusual at first sight. However, we want to provide the facility to abstract from concrete implementations. This means abstracting from concrete values as well. Suppose we have a concept Size that extracts the file size from a given file f. Furthermore, assume two functions `intSize` and `strSize` that, when applied to a file f, return the size of f as an integer or string value, respectively. These functions can be used for context definitions $Int = \texttt{intSize}(x)$ and $Str = \texttt{strSize}(x)$. Whenever we do not care whether the file size is represented by $\texttt{intSize}(f) = 12345$ or by $\texttt{strSize}(f) =$"12345", we can simply use the concept term $\mathsf{Size}(f)$ instead of $\mathsf{Size}(f)[Int]$ or $\mathsf{Size}(f)[Str]$, respectively.

In App. B.2, page 186, we derive concept semantics for sample concept expressions related to Name and EntryPoint. In the following we will often abbreviate $\mathcal{K}(t_1,...,t_n)[\_]$ by $\mathcal{K}(t_1,...,t_n)$. We conclude the technical introduction of concepts and contexts by proving well-definedness of the term structure $KT_{X,\Sigma,KD}$ for concept terms. Also, we

show that wildcard-free concept terms evaluate to a unique value.

Lemma 4.3.3 (Concept terms well-defined) Given a specification $Spec := (\Sigma, Sen)$, $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$, such that *Spec* is a permissible extension of the basic DA. Given furthermore a set TV of type variables suitable for Σ, a set of variables X suitable for Σ, a set R of role names suitable for TV, Σ, a set $KD \subseteq KD(R, TV, \Sigma)$, a Σ-model $\mathcal{A} \in A(\Sigma)$ for Spec, and a variable assignment $\eta \in Env(X, \mathcal{A})$. Then $KT_{X,\Sigma,KD}$ is a well-defined term structure and $\overset{t_d}{\rightsquigarrow}$ is a function for concept terms that do not contain _, i.e.,

(1) For all $\tau, \tau' \in \mathcal{T}$ it is true that $\tau < \tau'$ implies $KT_\tau(X, \Sigma, KD) \subseteq KT_{\tau'}(X, \Sigma, KD)$.

(2) For all types $\tau \in \mathcal{T}$ and all terms in $KT_\tau(X, \Sigma, KD)$ there is $v \in \tau^{\mathcal{A}} \cup \{\bot\}$ such that $(\mathcal{A}, \eta, t) \overset{t_d}{\rightsquigarrow} v$.

(3) For all types $\tau \in \mathcal{T}$ and all terms $\mathcal{K}(t_1, ..., t_n)[C]$ in $KT_\tau(X, \Sigma, KD)$, $C \neq \_$, there is a unique $v \in \tau^{\mathcal{A}} \cup \{\bot\}$ such that $(\mathcal{A}, \eta, t) \overset{t_d}{\rightsquigarrow} v$.

□

The formal arguments of the proof in App. C.2, page 214, have already been stated informally throughout this section. In particular, uniqueness of role assignments is important. Lemma 4.3.3(3) does *not* hold in presence of a wildcard as several values may be derivable due to the existential quantification in the semantics of Tab. 4.6. We close this chapter with a short summary and then move on to preservation in the next chapter.

4.4 Summary

We have introduced formal concept definitions. They include an *interface* and implementations (*context definitions*). We have distinguished non-functional and functional concepts. The former capture semantic relationships; the latter determine concrete values and, hence, can be used for extracting object-content. We have provided an example for each variant.

Concept expressions, i.e., *concept terms* and *concept formulas*, are used like regular terms and predicates, respectively. They, however, refer to concept implementations. By referring to implementations *by name* (or using wildcards), concept expressions hide implementation details. This keeps them readable.

We evaluate concept expressions in our dynamic environment where objects might not exists. For this purpose, we have defined *dynamic term values* and a notion of *dynamic formula satisfaction*. Two *correspondence theorems* show how static term values and static formula satisfaction carry over to the dynamic setting. In particular, the dynamic and static value of a given term are equal if all objects in the object-valued of that term exist.

There are important applications of these theorems. First, properties of digital objects can be proved in the static setting (e.g., using automated theorem provers). The correspondence theorems state under which circumstances these properties hold in the dynamic setting as well. Second, implications among concepts can be determined in the static setting. The correspondence theorems can be used to decide whether these

implications hold in the dynamic setting as well. As concept expressions are an integral part of the preservation language (see next chapter), this can help to avoid redundancies when specifying preservation requirements. As a result, the evaluation processes is potentially accelerated when migrations take place.

Chapter 5

Specifying and Evaluating Preservation Requirements

Up to now, we have formulated preservation requirements semi-formally. However, they cannot be processed in an automated way unless they are specified using a formal language with well-defined semantics. Therefore, we introduce a *preservation language*, which has a sound, state-based semantics. Preservation requirements are expressed w.r.t. concepts and are evaluated w.r.t. migration processes that run in the dynamic environment that has been introduced before. This integration into one combined method together with the coherently formal underpinning is the major contribution of this thesis. The road map is as follows:

- In Sect. 5.1 we give an informal overview. We particularly describe the co-action of the the preservation language and the single formal components that have been introduced before.
- A first notion of preservation is introduced in Sect. 5.2. It does not take into account object histories but relates concrete source and target objects to concepts.
- Object traces are introduced in Sect. 5.3. They are extracted from migration sequences and represent transformation paths of concrete objects.
- In Sect. 5.4 we combine object traces and the notion of object preservation of Sect. 5.1. We introduce transformation and preservation constraints as *trace formulas* and define their semantics w.r.t. traces. After that, we define a *preservation language*. This language includes quantified trace formulas (by means of trace quantifiers $\exists$ and $\forall$) in analogy to the well-known path quantifiers in temporal logics. Also, the language allows to specify preservation requirements for object *collections* conveniently. Semantically, preservation formulas are evaluated w.r.t. migration sequences and object traces that are extracted therefrom.

5.1 Informal Overview

Since this chapter rounds off our formal framework, we start with a brief explanation of how the preservation language integrates into the formal dynamic environment that has

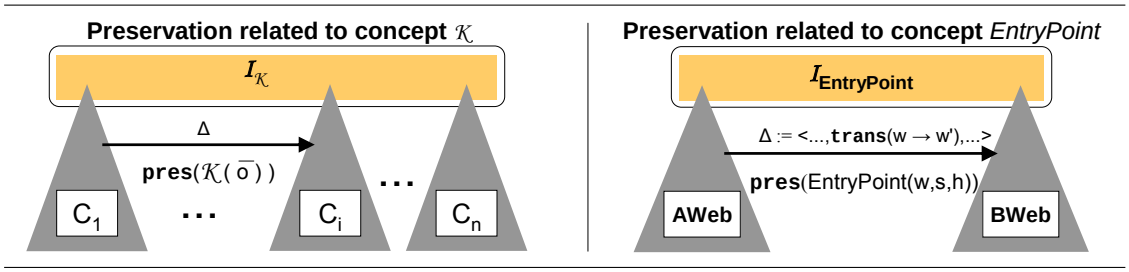


Figure 5.1: Preservation related to concepts

been introduced in the previous chapters. Fig. 5.1 shows how formal concept specifications, migration processes, and formal preservation constraints correlate. The left-hand part shows an abstract scheme for the preservation of a concept $\mathcal{K}$. $\mathcal{I}_{\mathcal{K}}$ denotes the interface of $\mathcal{K}$, $C_1, ..., C_n$ denote the implementing contexts, and Δ denotes a running migration process that has to adhere to a preservation requirement $\texttt{pres}(\mathcal{K}(\overline{o_i}))$. This scheme is instantiated on the right-hand side of Fig. 5.1 using the concept EntryPoint. Recall the semi-formal formulation of the preservation requirement for EntryPoint (Sect. 2.3):

> When transforming an object, which is assigned to one of the roles **website**, **source dir**, or **home page** of the concept EntryPoint, **all new versions** of all interface objects of EntryPoint must **satisfy the concept** EntryPoint **in** the **target context** *BWeb*, **if and only if** the **source objects satisfied** EntryPoint **in** the **source context** *AWeb*.

In Fig. 5.1, this requirement is represented by the constraint $\texttt{pres}(\mathsf{EntryPoint}(w, s, h))$, which is an abbreviated variant of those constraints that will be introduced in this chapter. It expresses the above requirement formally. In particular, it refers to a concrete concept (EntryPoint) and concrete source objects w, s, h. The constraint does not include the source and target context *AWeb* and *BWeb* for brevity as they are shown directly next to it in Fig. 5.1. Recall that our notion of preservation is related to *object histories*. The requirement above demands *new versions* of the source objects to satisfy EntryPoint. In our system object histories are traced using the distinguished state change $\mathsf{trans}(src \mapsto trg)$. This is indicated by the right-hand migration sequence in Fig. 5.1, which contains a transformation $\mathsf{trans}(w \mapsto w')$ for the source website w. When evaluating adherence to given preservation requirements, our system extracts object transformations from the underlying migration sequence and checks the requirements w.r.t. the so-obtained object histories. This shows the general co-action between the formal components of our framework. The last part (extraction of object histories and evaluation of preservation requirements) is part of the semantics of the formal preservation language that we introduce in this chapter. This language includes preservation constraints as regular language elements and supports *non-linear* histories. Recall that non-linear object histories occur in our running example. There, directories are duplicated twice when websites are transformed as html content is separated from non-html content.

Since our notion of preservation is related to object versions, we need a means for addressing concrete new object versions. In Fig. 5.2 we use the concept Contains (containment in directories) as an example. The left-hand part shows the source website of

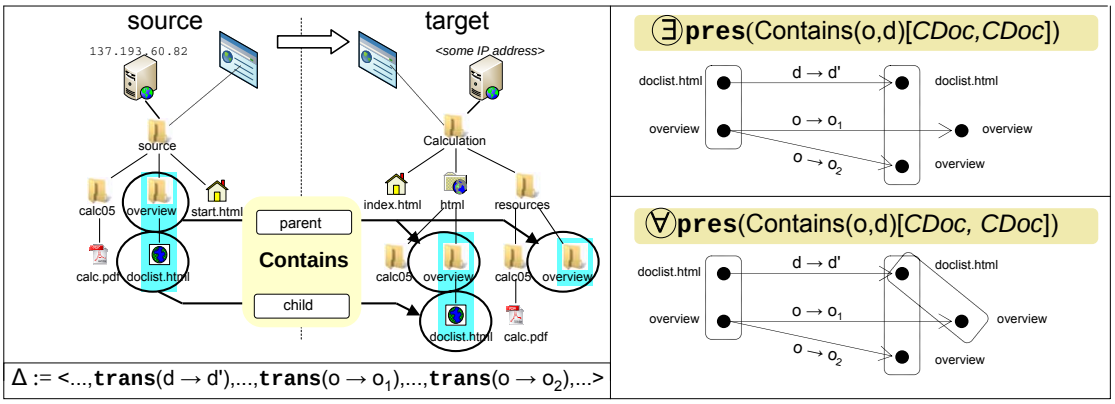


Figure 5.2: Preservation of concept Contains

the running example and a target website that respects the preservation requirements of Fig. 2.3, page 18. In particular, the source website is in format *AWeb*, and the target website is in format *BWeb*. The directory "overview" takes the role **parent**, the html file "doclist.html" takes the role **child**. The target website includes two new versions of the "overview" directory (i.e., this directory has a non-linear object history), but only one version of the html file. The relevant basic state transitions are shown in the bottom left-hand part of Fig. 5.2. Also, we observe that the containment concept does not hold for both "overview" directories and the html file in the target context. Hence, we need a means for expressing preservation requirements for *selected* paths of object histories.

As shown in the right-hand part, we select object versions by *trace quantifiers* ∃, ∀. This is adapted from temporal logics like *CTL* or *CTL** which are used in model checking [MOSS99]. While CTL includes temporal operators line *next* or *until*, we use

- *non-functional preservation constraints* of the form

$$\texttt{pres}_{nf}(\mathsf{Contains}(o, d)[CDoc, CDoc]),$$

- *functional preservation constraints* of the form $\texttt{pres}_{f}(\mathsf{Name}(w)[WebN, WebN])$, and
- *transformation constraints* of the form $d \mapsto \texttt{HTMLDoc}$

as atomic *trace formulas*. These formulas are evaluated w.r.t. traces instead of states, where leading trace quantifiers specify which traces to select. Transformation constraints enforce object transformations. Non-functional and functional preservation constraints enforce the preservation of non-functional and functional concepts, respectively. They are used identically and even hide whether the related concept is functional or not. We annotate them by $_{nf}$ and $_{f}$ for convenience only.

On the right-hand side of Fig. 5.2 we show the object histories for "doclist.html" and "overview". Versions are connected by arrows. The arrows are annotated with those transformations that have been extracted from the migration process Δ. Concept matchings are encircled. A trace is *one* parallel path of maximal length through the histories of all existing objects. Since traces have maximal length, they cover full object histories. Assuming $\mathcal{A}_s$ is the source state of Δ, $\mathcal{A}_r$ is its result state, and only *o*,*d* exist before Δ is executed, the (branching) object history shown in Fig. 5.2 is represented by two *separate* traces:

$$tr_1 := \begin{pmatrix} (\mathcal{A}_s, d) \longrightarrow (\mathcal{A}_r, d') \\ (\mathcal{A}_s, o) \longrightarrow (\mathcal{A}_r, o'_1) \end{pmatrix} \text{ and } tr_2 := \begin{pmatrix} (\mathcal{A}_s, d) \longrightarrow (\mathcal{A}_r, d') \\ (\mathcal{A}_s, o) \longrightarrow (\mathcal{A}_r, o'_2) \end{pmatrix}$$

Traces include combinations of a system state and an object. This directly relates object versions to system states, which is important when evaluating preservation constraints. The constraint

$$\mathtt{pres}_{nf}(\mathsf{Contains}(o, d)[\mathit{CDoc}, \mathit{CDoc}]),$$

e.g., holds in tr_1, if $\mathsf{Contains}(o, d)[\mathit{CDoc}]$ holds in $\mathcal{A}_s$ and $\mathsf{Contains}(o'_1, d')[\mathit{CDoc}]$ holds in $\mathcal{A}_t$. There, o'_1 and d' are new versions of o and d, respectively, and the assigned states $\mathcal{A}_s$ and $\mathcal{A}_t$ are necessary to evaluate the concept formulas. This underlines the strong correlation between concepts, contexts, and preservation.

The above-mentioned trace quantifiers now specify whether given preservation requirements must hold for at least one trace ($\textcircled{\exists}$) or for all traces ($\textcircled{\forall}$). The effect of both trace quantifiers is illustrated in the right-hand part of Fig. 5.2. The upper part shows the semantics of an existential preservation constraint for the Contains concept. It suffices to find at least one trace in $\{tr_1, tr_2\}$ such that the new versions of the source objects match Contains in *CDoc*. In contrast, the $\textcircled{\forall}$ variant requires this property to hold for both tr_1 and tr_2.

The specification language explained so far still lacks usability. Since DAs usually host large object collections, the preservation language must support (1) selecting object collections and (2) connecting concrete objects of these collections to preservation requirements. We implement this by combining suitable FOPL formulas and trace formulas. The *preservation formula*

$$\begin{array}{l} \forall w : \mathtt{Website} \bullet \\ \quad \forall d, d' : \mathtt{Dir} \bullet \\ \qquad \mathsf{WebSrc}(d, w) \wedge \mathsf{Contains}(d, d')[\mathit{CDirRec}] \Rightarrow \\ \qquad\quad \textcircled{\exists}\ (d \mapsto \mathtt{Dir} \wedge d' \mapsto \mathtt{Dir} \wedge \mathtt{pres}_{nf}(\mathsf{Contains}(d, d')[\mathit{CDirRec}, \mathit{CDirHtml}])) \wedge \\ \qquad\quad \textcircled{\exists}\ (d \mapsto \mathtt{Dir} \wedge d' \mapsto \mathtt{Dir} \wedge \mathtt{pres}_{nf}(\mathsf{Contains}(d, d')[\mathit{CDirRec}, \mathit{CDirRes}])) \end{array}$$

indicates the expressiveness of the full preservation language. There, some contexts and concepts occur that have not been introduced before. Therefore, we explicitly explain the semantics of this formula:

- For any website w and directories d, d' ($\forall w : \mathtt{Website}, d, d' : \mathtt{Dir}$)
- such that d is the source directory of w ($\mathsf{WebSrc}(d, w)$) and the new version of d' is a sub-directory of the new version of d ($\mathsf{Contains}(d, d')[\mathit{CDirRec}]$)
- there is a trace such that d and d' are transformed to type Dir and the new version of d' is a sub-directory of the new version of d in the context *CDirHtml* ($\textcircled{\exists}\ d \mapsto \mathtt{Dir} \wedge d' \mapsto \mathtt{Dir} \wedge \mathtt{pres}_{nf}(\mathsf{Contains}(d, d')[\mathit{CDirRec}, \mathit{CDirHtml}])$), and
- there is a trace such that d and d' are transformed to type Dir and d' is a sub-directory of d in the context *CDirRes* ($\textcircled{\exists}\ d \mapsto \mathtt{Dir} \wedge d' \mapsto \mathtt{Dir} \wedge \mathtt{pres}_{nf}(\mathsf{Contains}(d, d')[\mathit{CDirRec}, \mathit{CDirRes}])$)

Recall that html content and non-html content are separated in *BWeb*. For this purpose, the source directory of the target website directly contains directories "html" and

"resources", respectively, which is shown in Fig. 5.2. Since "resources" may contain non-html content only and "html" may contain html content only, Contains is implemented in two separate contexts *CDirRes* and *CDirHtml*, respectively. These contexts occur in the preservation formula above. When setting w ="Calculation" (example website), d ="source" (source directory of w), and d' ="overview", this preservation formula requires that

- both "source" and "overview" are transformed and
- the sub-directory relationship is maintained, but once for the "html" directory and once for the "resources" directory.

The transformation result in Fig. 5.2 satisfies these requirements. As an example, two versions have been created for the directory "overview". Both are sub-directories of the directory "Calculation", but one resides in "html", and one resides in "resources".

The preservation formula above shows two important properties of the preservation language:

(1) Concept terms and concept formulas are fully integrated

(2) Preservation constraints are evaluated w.r.t. given concepts where concepts and the source and target context are referred to *by name*.

This supports re-use and users can express preservation requirements in an implementation-independent way.

5.2 Preservation — a First Account

Our first notion of preservation relates concepts and source and target objects, respectively, that match the concepts' implementations. It will be the basis for the following sections, where we supplement this notion by object traces and introduce the formal semantics of the preservation language.

In general, formal notions of preservation can be found in different variations throughout the relevant literature [MCG05, Por05, Grz97, WO00, ADK06]. We postpone a discussion to Chap. 14, but show an abstract preservation scheme in Fig. 5.3 that covers most of these variants. According to this scheme, preservation means to preserve a model property under abstractions a, a'. When transforming a source model to a target model (indicated by δ), the abstraction result of respective parts of the source model (using a) must correspond to the abstraction result (using a') of the transformation target. Correspondence is denoted by $\approx$ in Fig. 5.3. Clearly, the real effect of this notion depends on the chosen abstraction functions and the $\approx$ operator. In its left-hand and right-hand part, Fig. 5.3 shows two instantiations of the scheme. They correspond to non-functional and functional concept preservation; these two variants are supported by our preservation language.

Since we consider migration sequences, we have substituted δ by Δ. Concept interfaces determine those parts of the source and target model that need to be related by the notion of preservation. Concerning non-functional concept preservation, abstraction

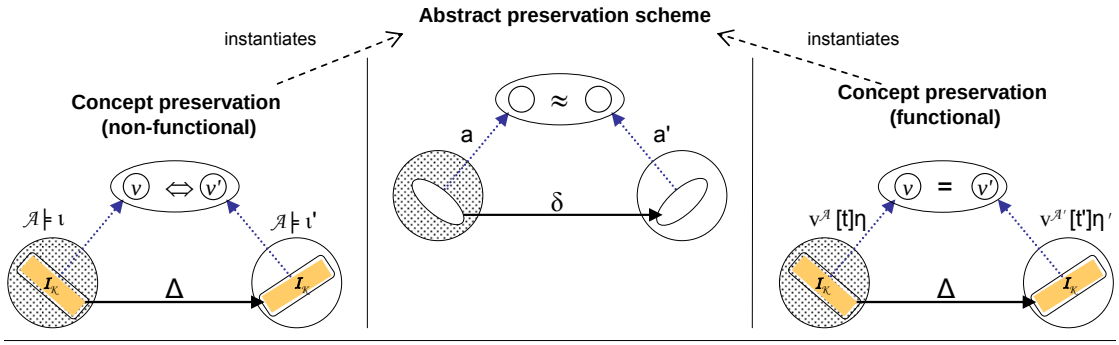


Figure 5.3: Abstract preservation scheme and instantiations with non-functional and functional concept preservation

is done by the implementing formula ι and ι' for the source and target context, respectively. Hence, the interface objects are abstracted to the respective truth value. Only if both formulas return *equivalent truth values*, $\mathcal{K}$ is preserved .

Preservation of functional concepts works quite similar. Abstraction, however, is determined by the implementing *terms* t and t' of the source and target context, respectively. The $\approx$-operator is instantiated by equality. Only if both terms return *equal values*, $\mathcal{K}$ is preserved.

Clearly, other instantiations of the preservation scheme may be useful depending on the application domain. We shall see later on, that our preservation language can be extended to other variants with little effort.

In the following definition we introduce a first notion of preservation, which directly implements the just-described variants of preservation.

Definition 5.2.1 (Preservation predicate) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA, two Σ-models $\mathcal{A}_s, \mathcal{A}_t$ for $Spec$, a set TV of type variables suitable for Σ, a set of variables X suitable for Σ, a set R of role names suitable for TV, Σ, a set $KD \subseteq KD(R, TV, \Sigma)$, two concept expressions $\mathcal{K}(s_1, ..., s_n)[C_s], \mathcal{K}(t_1, ..., t_n)[C_t] \in KT_{\mathtt{Top}}(X, \Sigma, KD) \cup KF(X, \Sigma, KD)$ and two variable assignments η_s, η_t. Then $\mathcal{K}$ *is preserved for* $\overline{s_i}, C_s, \mathcal{A}_s, \eta_s$ *in* $\overline{t_i}, C_t, \mathcal{A}_t, \eta_t$ iff

$$(\mathcal{A}_s, \mathcal{A}_t) \models_d pres(\mathcal{K}(s_1, ...s_n)[C_s], \mathcal{K}(t_1, ...t_n)[C_t])[\eta_s, \eta_t],$$

where

$$(\mathcal{A}_s, \mathcal{A}_t) \models_d pres(\mathcal{K}(s_1, ...s_n)[C_s], \mathcal{K}(t_1, ...t_n)[C_t])[\eta_s, \eta_t] \Leftrightarrow$$
$$\begin{cases} \exists v \in \mathtt{Top}^{\mathcal{A}_s} \bullet (\mathcal{A}_s, \eta_s, \mathcal{K}(s_1, ...s_n)[C_s]) \overset{t_d}{\rightsquigarrow} v \ \wedge (\mathcal{A}_t, \eta_t, \mathcal{K}(t_1, ...t_n)[C_t]) \overset{t_d}{\rightsquigarrow} v, & \mathcal{K} \text{ funct.} \\ \mathcal{A}_s \models_d \mathcal{K}(s_1, ...s_n)[C_s][\eta_s] \Leftrightarrow \mathcal{A}_t \models_d \mathcal{K}(t_1, ...t_n)[C_t][\eta_t], & \mathcal{K} \text{ non-funct.} \end{cases}$$

□

Preservation is modeled w.r.t. concrete source and target objects and w.r.t. a given concept $\mathcal{K}$. By using the dynamic semantics $\models_d$, we implicitly require the source and target objects $s_1, ..., s_n$ and $t_1, ..., t_n$ to exist in the source and target state, respectively; "preservation" makes sense for existing objects only. Notice that putting together concept terms $KT_{\mathtt{Top}}(X, \Sigma, KD)$ and concept formulas $KF(X, \Sigma, KD)$ to concept expressions in

the definition above does not lead to ambiguity. Concept definitions are not-overloaded. Hence,

$$\mathcal{K}(s_1,...,s_n)[C_s] \in KT_\tau(X,\Sigma,KD) \text{ iff } \mathcal{K}(t_1,...,t_n)[C_t] \in KT_\tau(X,\Sigma,KD)$$

and

$$\mathcal{K}(s_1,...,s_n)[C_s] \in KF(X,\Sigma,KD) \text{ iff } \mathcal{K}(t_1,...,t_n)[C_t] \in KF(X,\Sigma,KD).$$

Concerning functional concepts abstraction as shown in Fig. 5.3 is instantiated by the value relation $\overset{t_d}{\rightsquigarrow}$. $\mathcal{K}$ is preserved for $s_1,...,s_n$ in $t_1,...,t_n$ if there is a value v such that $(\mathcal{A}_s,\eta_s,\mathcal{K}(s_1,...s_n)[C_s]) \overset{t_d}{\rightsquigarrow} v$ and $(\mathcal{A}_t,\eta_t,\mathcal{K}(t_1,...t_n)[C_t]) \overset{t_d}{\rightsquigarrow} v$. C_s and C_t may be wildcards. If $\mathcal{K}$ is non-functional, formula validity is used as abstraction. A non-functional $\mathcal{K}$ is preserved if it holds for the source objects $s_1,...,s_n$ in the source context C_s if and only if it holds for the target objects $t_1,...,t_n$ in the target context C_t. There, we deliberately require the strong correspondence "if and only if" — we preserve the *status* of concept satisfaction. As a consequence, a status change of the form "if the concept does not hold in the source context, it is to hold in the target context" can be enforced using a negated constraint. If we required

$$\mathcal{A}_s \models_d \mathcal{K}(s_1,...s_n)[C_s][\eta_s] \Rightarrow \mathcal{A}_t \models_d \mathcal{K}(t_1,...t_n)[C_t][\eta_t]$$

in the definition above, this could not easily be expressed. In App. B.3, page 187, we list samples for the semantics of *pres*.

Defn. 5.2.1 does not yet include migration sequences and, thus, object histories. It will, however, be used in the semantics for trace formulas later on.

According to the following lemma the *pres* relation works sequentially.

Lemma 5.2.1 (Preservation is sequential) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA, three Σ-models $\mathcal{A},\mathcal{A}',\mathcal{A}''$ for $Spec$, a set TV of type variables suitable for Σ, a set of variables X suitable for Σ, a set R of role names suitable for TV,Σ, a set $KD \subseteq KD(R,TV,\Sigma)$, three concept expressions $\mathcal{K}(t_1,...,t_n)[C]$, $\mathcal{K}(t'_1,...,t'_n)[C']$, $\mathcal{K}(t''_1,...,t''_n)[C'']$ in $KT_{\mathtt{Top}}(X,\Sigma,KD) \cup KF(X,\Sigma,KD)$ such that $C' \neq \_$, and three variable assignments η,η',η''. Then

$$\begin{array}{c}(\mathcal{A},\mathcal{A}') \models pres(\mathcal{K}(t_1,...t_n)[C],\mathcal{K}(t'_1,...t'_n)[C'])[\eta,\eta'] \wedge \\ (\mathcal{A}',\mathcal{A}'') \models pres(\mathcal{K}(t'_1,...t'_n)[C'],\mathcal{K}(t''_1,...t''_n)[C''])[\eta',\eta''] \\ \Rightarrow \\ (\mathcal{A},\mathcal{A}'') \models pres(\mathcal{K}(t_1,...t_n)[C],\mathcal{K}(t''_1,...t''_n)[C''])[\eta,\eta'']\end{array}$$

□

The proof can be found in App. C.3, page 215. Notice that the prerequisite $C' \neq \_$ is necessary. The semantics for concept terms and concept formulas with wildcards only requires a context to *exist*. This is, in general, insufficient for proving the just-shown sequential behavior.

Later on, we will see that this sequential behavior carries over to the composition of migration processes. If two single processes preserve a concept, the process that is the result of concatenating these processes does as well. In other words, global preservation

follows from iterated single-step-preservation. The sequential behavior of the *pres* relation is a must in digital archiving. Local (w.r.t. time) migration decisions have to be made with only uncertain information of how technologies evolve. Hence, it is important to know that a property that is preserved by two sequential migration steps is preserved globally.

5.3 Object Traces

The *pres*-predicate does not yet consider object *histories*; it requires the source and target objects to satisfy the same concept only. Our desired notion of preservation, however, is strongly related to object histories; the target objects must be new *versions* of the source objects. Recall that we bind the notion of object histories to basic transitions $\mathsf{trans}(t_{src} \mapsto t_{trg})$. This has two advantages. First, we assume as little as necessary about the basic DA. Second, requiring object histories to be part of the DA may cause problems when deleting objects. Suppose we want the name of a website w to be preserved by a transformation process Δ. Assume $src(\Delta) = \mathcal{A}$ and $res(\Delta) = \mathcal{A}'$ and w is deleted during this process (i.e., there is an operation $\mathsf{del}(w) \in \Delta$). If the notion of object histories relied on an implementation in *DA*s, only, (e.g., using a dynamic function `history`), we could not relate w to its new version w'. The reason is that we try to relate w and w' *semantically* in a state $\mathcal{A}'$. Yet w does not exist anymore in $\mathcal{A}'$. We avoid this by tracing object histories *syntactically* using explicit transformation operations $\mathsf{trans}(w \mapsto w')$.

Formal object traces are defined as follows.

Definition 5.3.1 (Object traces) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA and a migration sequence Δ over Σ such that $src(\Delta)$ is derived by basic transitions. Then *the set* $traces_i(\Delta)$ *of initial traces over* Δ *of length* i $(0 \leq i)$ is defined inductively as follows:

$$
\begin{array}{lll}
traces_0(\Delta) & := \{ & \langle (src(\Delta), t) \rangle \mid t \in GT_{\mathtt{DObj}}(\Sigma), src(\Delta) \models_d t \in \mathtt{existDObj}, \} \\
traces_{i+1}(\Delta) & := & traces_i(\Delta) \fatsemi step(\Delta), \text{ where} \\
 & & \\
step(\Delta) & := \{ & \langle (\mathcal{A}_0, t_0), (\mathcal{A}_1, t_1) \rangle \mid \langle \mathcal{A}_0, \mathsf{trans}(t_0 \mapsto t_1), \mathcal{A}_1 \rangle \subseteq \Delta \} \\
tr \fatsemi tr' & := \{ & \langle (\mathcal{A}_0, t_0) ..., (\mathcal{A}_{n-1}, t_{n-1}), (\mathcal{A}'_0, t'_0), (\mathcal{A}'_1, t'_1) \rangle \mid \\
 & & \langle (\mathcal{A}_0, t_0), ..., (\mathcal{A}_{n-1}, t_{n-1}), (\mathcal{A}_n, t_n) \rangle \in tr, \\
 & & \langle (\mathcal{A}'_0, t'_0), (\mathcal{A}'_1, t'_1) \rangle \in tr', \mathcal{A}'_0 \models_d t_n = t'_0 \}
\end{array}
$$

Given a trace $tr := \langle (\mathcal{A}_0, t_0), ..., (\mathcal{A}_n, t_n) \rangle \in traces_n(\Delta)$ and $0 \leq j \leq n$, we define

$$
\begin{array}{lll}
|tr| & := n & (\textit{trace length}) \\
version(j, tr) & := t_j & (j\textit{-th version of } t_0 \textit{ in } tr) \\
state(j, tr) & := \mathcal{A}_j & (j\textit{-th state of } tr) \\
src(tr) & := state(0, tr) & (\textit{trace source}) \\
res(tr) & := state(|tr|, tr) & (\textit{trace result})
\end{array}
$$

The set $maxtraces(\Delta)$ *of all inclusionmaximal traces over* Δ and *the set* $maxtraces(\Delta, t)$ *of inclusionmaximal traces over* Δ *for* t, $t \in GT_{\mathtt{DObj}}(\Sigma)$, are defined as follows:

$$\begin{aligned} &\mathit{maxtraces}(\Delta) := \\ &\quad \bigcup_{i\in\mathbb{N}} \{\langle(\mathit{src}(\Delta), t_0), ..., (\mathit{res}(\Delta), t_i)\rangle \mid \langle(\mathcal{A}_0, t_0), ..., (\mathcal{A}_i, t_i)\rangle \in \mathit{traces}_i(\Delta), \\ &\qquad\qquad \langle(\mathcal{A}_0, t_0), ..., (\mathcal{A}_i, t_i)\rangle \fatsemi \mathit{step}(\Delta) = \emptyset, \\ &\qquad\qquad \mathit{res}(\Delta) \models_d t_i \in \texttt{existDObj}\} \end{aligned}$$

$$\mathit{maxtraces}(\Delta, t) := \{tr \mid tr \in \mathit{maxtraces}(\Delta), \mathit{state}(0, tr) \models_d \mathit{version}(0, tr) = t\}$$

The set $\mathcal{TR}(\Delta)$ of *full traces of* Δ is defined by

$$\mathcal{TR}(\Delta) := \bigotimes_{\substack{t\in GT_{\texttt{DObj}}(\Sigma) \\ \mathit{src}(\Delta)\models_d t\in\texttt{existDObj}}} \mathit{maxtraces}(\Delta, t),$$

where the projections are denoted by

$$\pi_t : \mathcal{TR}(\Delta) \to \mathit{maxtraces}(\Delta, t)\ (t \in GT_{\texttt{DObj}}(\Sigma), \mathit{src}(\Delta) \models_d t \in \texttt{existDObj}).$$

Given a trace tuple $tr \in \mathcal{TR}(\Delta)$, we set

$$\begin{aligned} \mathit{src}(tr) &:= \mathit{src}(\Delta)\ (\textit{trace source}) \\ \mathit{res}(tr) &:= \mathit{res}(\Delta)\ (\textit{trace result}) \end{aligned}$$

□

Traces contain tuples that consist of a system state and an object. That way we relate concrete system states to object histories, which allows for expressive trace properties later on. Apart from that, we distinguish *initial* traces and *inclusionmaximal* traces. The trace source of all initial traces exists in $\mathit{src}(\Delta)$. Initial traces of length zero cover all those objects that exist in the source state $\mathit{src}(\Delta)$ — objects are versions of themselves. The set $\mathit{step}(\Delta)$ is equivalent to those subsequences of Δ that contain a single transformation step $\langle\mathcal{A}_0, \mathsf{trans}(t_0 \mapsto t_1), \mathcal{A}_1\rangle$. By concatenating traces_j and $\mathit{step}(\Delta)$ using $\fatsemi$, traces of length $j+1$ are generated. According to the definition of $\fatsemi$, the result object of $tr \in \mathit{traces}_j(\Delta)$ must equal the source object of $tr' \in \mathit{step}(\Delta)$ in order for $\fatsemi$ to be applicable. Hence, objects in those traces generated by $\fatsemi$ are always versions of the respective source object. As this does not yet assure that trace results exist in the result state of Δ, we explicitly require this in the definition of $\mathit{maxtraces}(\Delta)$. This is no limitation as traces ending in non-existing target objects are useless for our purposes — non-existing objects cannot preserve anything. In the semantics for preservation constraints we will use inclusionmaximal traces only. These traces

(1) start in $\mathit{src}(\Delta)$,
(2) end in $\mathit{res}(\Delta)$, and
(3) contain history branches of their source objects of maximal length.

In particular, maximal traces for concrete objects $\mathit{maxtraces}(\Delta, t)$ will be important later on. They can be extracted from *full parallel* traces in $\mathcal{TR}(\Delta)$. $\mathcal{TR}(\Delta)$ includes all *tuples* of inclusionmaximal traces of those objects that exist in $\mathit{src}(\Delta)$ and will be used in the semantics of preservation constraints. The projections to $\mathit{maxtraces}(\Delta, t)$ are denoted by π_t. The functions *src* and *res* are extended to trace tuples as well. In analogy to

inclusionmaximal traces, the source and target of a trace tuple yields the source and target state of the corresponding migration sequence. In App. B.3, page 189, we list sample traces for the example migration shown in the introduction to this chapter.

In Defn. 5.3.1 we define functions that can be used to refer to the components of traces conveniently. Since term t_i is the result of transforming t_0 i times, we call it the i-th version of the trace tr, which is denoted by $t_i = \mathit{version}(i, tr)$. Analogously, the i-th state is identified by $\mathit{state}(i, tr)$. These functions fully determine traces. For example,

$$tr := \langle(\mathcal{A}_0, t_0), (\mathcal{A}_1, t_1), ..., (\mathcal{A}_n, t_n)\rangle$$

can also be written as

$$\langle(\mathit{state}(0, tr), \mathit{version}(0, tr)), (\mathit{state}(1, tr), \mathit{version}(1, tr)), ...,$$
$$(\mathit{state}(n, tr), \mathit{version}(n, tr))\rangle.$$

The following lemma states that (1) $\mathit{version}(i, tr)$ exists in $\mathit{state}(i, tr)$ for all i,tr and (2) traces are circle-free. Object histories can, thus, be understood as trees.

Lemma 5.3.1 (Object versions exist, traces are circle free) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA, a migration sequence Δ over Σ such that $\mathit{src}(\Delta)$ is derived by basic transitions, and an initial trace $tr \in \mathit{traces}_n(\Delta)$ of length n. Then

(1) $\mathit{state}(i, tr) \models_d \mathit{version}(i, tr) \in \texttt{existDObj}$ for all $0 \leq i \leq n$ and

(2) the trace is circle-free, i.e., $\mathit{state}(j, tr) \models \texttt{oid}(\mathit{version}(i, tr)) \neq \texttt{oid}(\mathit{version}(j, tr))$ for all $1 \leq i < j \leq n$.

□

The proof can be found in App. C.3, page 215. It exploits the fact that the source state of Δ is derived by basic transitions and, thus, has been generated starting at an initial state where no objects exist (cf. Defn. 3.3.9, page 51). As Δ is a valid migration sequence, intermediate system sates are subsequent to each other w.r.t. one of the basic transitions (Defn. 3.3.8). Hence, the system invariants of Cor. 3.3.1 hold; this guarantees circle-freeness.

5.4 Preservation Formulas – Relating Preservation and Object Traces

Using the basic preservation predicate *pres* and object traces we can define the semantics of *transformation* and *preservation constraints*. They belong to the class of *trace formulas*.

Definition 5.4.1 (Trace formulas) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA, a set TV of type variables suitable for Σ, a set of variables X suitable for Σ, a set R of role names suitable for TV, Σ, and a set $KD \subseteq KD(R, TV, \Sigma)$. Then the *formula structure* $TF_{X,\Sigma,KD}$ *of trace formulas over* X, Σ, KD is defined by

$$TF_{X,\Sigma,KD} := FOL((TF^{at}(X, \Sigma, KD), \bigcup_{\Delta \in MS(\Sigma)} \mathcal{TR}(\Delta), \models_d, \mathcal{FV}), dom),$$

Table 5.1: *Syntax and semantics of atomic trace formulas*

Syntax:

(1) **Transformation constraints:**

$$\frac{t \in T_{\texttt{DObj}}(X,\Sigma)\ \tau \in \mathcal{T}, \tau{<}\texttt{DObj}}{t \mapsto \tau \in TF^{at}(X,\Sigma,KD)} \qquad \mathcal{FV}(t \mapsto \tau) = \mathcal{FV}(t)$$

(2) **Functional concept preservation constraints:**

$$\frac{\begin{array}{l}\mathcal{K}\ \mathcal{I} = \{C_1 = \iota_{C_1}, ..., C_m = \iota_{C_m}\} \in KD \\ \mathcal{K}(t_1,...,t_n)[C_s] \in KT(X,\Sigma,KD) \\ C_t \in \{C_1,...,C_m,\_\}\end{array}}{\texttt{pres}_f(\mathcal{K}(t_1,...,t_n)[C_s,C_t]) \in TF^{at}(X,\Sigma,KD)} \qquad \mathcal{FV}(\texttt{pres}_f(\mathcal{K}(t_1,...,t_n)[C_s,C_t])) = \bigcup_{1\le i\le n} \mathcal{FV}(t_i)$$

(3) **Non-functional concept preservation constraints:**

$$\frac{\begin{array}{l}\mathcal{K}\ \mathcal{I} = \{C_1 = \iota_{C_1}, ..., C_m = \iota_{C_m}\} \in KD \\ \mathcal{K}(t_1,...,t_n)[C_s] \in KF(X,\Sigma,KD) \\ C_t \in \{C_1,...,C_m,\_\}\end{array}}{\texttt{pres}_{nf}(\mathcal{K}(t_1,...,t_n)[C_s,C_t]) \in TF^{at}(X,\Sigma,KD)} \qquad \mathcal{FV}(\texttt{pres}_{nf}(\mathcal{K}(t_1,...,t_n)[C_s,C_t])) = \bigcup_{1\le i\le n} \mathcal{FV}(t_i)$$

Semantics:

(1) **Transformation constraints:**

$$\frac{\begin{array}{ll}t \mapsto \tau \in TF^{at}(X,\Sigma,KD) & tr \in \bigcup_{\Delta\in MS(\Sigma)} \mathcal{TR}(\Delta) \\ t' \equiv \mathit{version}(|\pi_t(tr)|, \pi_t(tr)) & \eta \in Env(X, src(tr)) \\ |\pi_t(tr)| \ge 1 & res(tr) \models_d \texttt{inst}_\tau(t')[\eta]\end{array}}{tr \models_d t \mapsto \tau[\eta]}$$

(2) **Functional concept preservation constraints:**

$$\frac{\begin{array}{ll}\texttt{pres}_f(\mathcal{K}(t_1,...,t_n)[C_s,C_t]) \in & tr \in \bigcup_{\Delta\in MS(\Sigma)} \mathcal{TR}(\Delta) \\ TF^{at}(X,\Sigma,KD) & \mathcal{A} = src(tr),\ \mathcal{A}' = res(tr) \\ t'_1 \equiv \mathit{version}(|\pi_{t_1}(tr)|, \pi_{t_1}(tr)), ..., & \eta \in Env(X,\mathcal{A}) \\ t'_n \equiv \mathit{version}(|\pi_{t_n}(tr)|, \pi_{t_n}(tr)) & (\mathcal{A},\mathcal{A}') \models_d pres(\mathcal{K}(t_1,...,t_n)[C_s], \mathcal{K}(t'_1,...,t'_n)[C_t])[\eta,\eta]\end{array}}{tr \models_d \texttt{pres}_f(\mathcal{K}(t_1,...,t_n)[C_s,C_t])[\eta]}$$

(3) **Non-functional concept preservation constraints:**

$$\frac{\begin{array}{ll}\texttt{pres}_{nf}(\mathcal{K}(t_1,...,t_n)[C_s,C_t]) \in & tr \in \bigcup_{\Delta\in MS(\Sigma)} \mathcal{TR}(\Delta) \\ TF^{at}(X,\Sigma,KD) & \mathcal{A} = src(tr),\ \mathcal{A}' = res(tr) \\ t'_1 \equiv \mathit{version}(|\pi_{t_1}(tr)|, \pi_{t_1}(tr)), ..., & \eta \in Env(X,\mathcal{A}) \\ t'_n \equiv \mathit{version}(|\pi_{t_n}(tr)|, \pi_{t_n}(tr)) & (\mathcal{A},\mathcal{A}') \models_d pres(\mathcal{K}(t_1,...,t_n)[C_s], \mathcal{K}(t'_1,...,t'_n)[C_t])[\eta,\eta]\end{array}}{tr \models_d \texttt{pres}_{nf}(\mathcal{K}(t_1,...,t_n)[C_s,C_t])[\eta]}$$

where the set $TF^{at}(X,\Sigma,KD)$ of *atomic trace formulas over* X,Σ,KD, the satisfaction relation $\models_d$, and the function $\mathcal{FV}$ are defined as shown in Tab. 5.1. Moreover,

$$dom(tr, x:\tau, \eta) := \begin{cases} (src(tr), \{v |\ v \in \tau^{src(tr)}, ct \in CT^{min}_{src(tr)}(\Sigma, v), \\ \qquad\qquad (src(tr), \eta, ct) \overset{t_d}{\rightsquigarrow} v\}) & ,\ \eta \in Env(X, src(tr)) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

for all $(tr, x:\tau, \eta) \in (\bigcup_{\Delta\in MS(\Sigma)} \mathcal{TR}(\Delta)) \times X \times Env(X,\Sigma)$.

In the following $TF(X,\Sigma,KD)$ (formulas), $\models_d$ (satisfaction relation), and $\mathcal{FV}$ (free variables) denote the respective component of $TF_{X,\Sigma,KD}$. □

Transformation constraints, functional, and non-functional concept preservation constraints are atomic trace formulas. Full trace formulas are first order formulas over atomic ones. Certainly, other kinds of trace formulas may be thinkable depending on the application domain. Typical examples comprise *next* or *until*-operators that are used in temporal logics for model checking dynamic systems. Since traces include intermediate

system states, these operators can be defined in a straightforward way. For our purposes, however, the just-defined trace formulas are sufficient.

In order for $t \mapsto \tau$ to be well-formed, the source object t must be of type `DObj` or below. The same is true for the target type τ. The semantics is determined w.r.t. a trace tuple $tr \in \mathcal{TR}(\Delta)$. It requires the *final* version of the trace $\pi_t(tr)$ for t to be of type τ. Also, the trace must have length one or greater. Hence, transformation constraints *enforce* transformations; $t \mapsto \tau$ does not hold before t is transformed appropriately. By using the *final* version in the semantics, we implement a design goal of our preservation language: We only constrain migration processes by *pre- and post conditions* and do not restrict intermediate steps of a process. As a result, we have no notion of sub-traces. Also, trace quantifiers $\textcircled{\exists}$ and $\textcircled{\forall}$ cannot be used recursively in contrast to other related languages like *CTL*. Again, our decision is strongly motivated by the application domain. The language can be extended if desired.

Syntax and semantics of concept preservation constraints is defined quite similar for functional and non-functional concepts. We annotate the constraints by $_f$ and $_{nf}$, respectively, for clarity. Concept preservation constraints $\texttt{pres}_f(\mathcal{K}(t_1, ..., t_n)[C_s, C_t])$ comprise a concept term $\mathcal{K}(t_1, ..., t_n)$, a source context C_s, and a target context C_t. C_s and C_t are used for matching $\mathcal{K}$ w.r.t. the source and target objects, respectively. Notice that C_s, C_t may be wildcards. If both are wildcards, we will sometimes omit them in the following.

The semantics of concept preservation constraints integrates preservation according to Defn. 5.2.1 into our dynamic environment. In particular, it relates the interface objects $t_1, ..., t_n$ and the trace results $t'_1, ..., t'_n$ of their traces $\pi_{t_1}(tr), ..., \pi_{t_n}(tr)$. The *pres* relation must hold for $\mathcal{K}(\overline{t_i})[C_s]$ and $\mathcal{K}(\overline{t'_i})[C_t]$ in $(\mathcal{A}, \mathcal{A}')$ under η. This is the desired notion of preservation as explained in the introduction to this chapter. $\mathcal{A}, \mathcal{A}'$ are the source and target state of the trace for t_1, respectively. As all traces for Δ start in $src(\Delta)$ and end in $res(\Delta)$, the choice of t_i is irrelevant. Also, the semantics is well-defined since all t'_i exist in $res(\Delta)$ according to Defn. 5.3.1(Traces).

The domain mapping *dom*, which is used to extend atomic trace formulas to first order logic, is similar to the one used for the dynamic semantics of Σ-formulas (cf. Defn. 4.3.2, page 63). However, trace formulas are evaluated w.r.t. trace tuples $tr \in \mathcal{TR}(\Delta)$. Given a trace formula $\forall x : \tau \bullet \phi$, we, thus, have to extract a suitable system state from tr in order to determine the quantifier sphere for $x : \tau$. Since preservation requirements are evaluated for *source* objects of traces we use $src(tr)$.

Continuative explanations on trace formulas can be found in App. B.3, page 189; according to the next lemma, trace formula semantics is well-defined.

Lemma 5.4.1 (Semantics of trace formulas is well-defined) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA, a set TV of type variables suitable for Σ, a set of variables X suitable for Σ, a set R of role names suitable for TV, Σ, and a set $KD \subseteq KD(R, TV, \Sigma)$. Then $TF_{X,\Sigma,KD}$ is well-defined, i.e.:

(1) For all $tr \in \bigcup_{\Delta \in MS(\Sigma)} \mathcal{TR}(\Delta)$, $x : \tau \in X$, and $\eta \in Env(X, \Sigma)$ it is true that $dom(tr, x : \tau, \eta) = (\mathcal{A}, d)$ implies $\eta \in Env(X, \mathcal{A})$ and $d \subseteq \tau^{\mathcal{A}}$.

□

Table 5.2: *Syntax and semantics of atomic preservation formulas*

Syntax:

(1) **Trace formulas:**

$$\frac{\phi \in TF(X,\Sigma,KD)}{\textcircled{\forall}\phi \in PF^{at}(X,\Sigma,KD)} \qquad \mathcal{FV}(\textcircled{\forall}\phi) = \mathcal{FV}(\phi)$$

(2) **Atomic formulas over Σ-terms and concept terms:**

$$\frac{(F, A(\Sigma), \models', \mathcal{FV}') = AT(KT_{X,\Sigma,KD} \uplus T^d_{X,\Sigma})\ \ \phi \in F}{\phi \in PF^{at}(X,\Sigma,KD)} \qquad \mathcal{FV}(\phi) = \mathcal{FV}'(\phi)$$

(3) **Concept formulas:**

$$\frac{\phi \in KF(X,\Sigma,KD)}{\phi \in PF^{at}(X,\Sigma,KD)}$$

Semantics:

(1) **Trace formulas:**

$$\frac{\textcircled{\forall}\phi \in PF^{at}(X,\Sigma,KD) \quad \forall tr \in \mathcal{TR}(\Delta)\ \bullet\ tr \models_d \phi[\eta] \quad \Delta \in MS(\Sigma)}{\Delta \models_d \textcircled{\forall}\phi[\eta]}$$

(2) **Atomic formulas over Σ-terms and concept terms:**

$$\frac{(F, A(\Sigma), \models', \mathcal{FV}') = AT(KT_{X,\Sigma,KD} \uplus T^d_{X,\Sigma}), \phi \in F \quad src(\Delta) \models' \phi[\eta] \quad \Delta \in MS(\Sigma)}{\Delta \models_d \phi[\eta]}$$

(3) **Concept formulas:**

$$\frac{\phi \in KF(X,\Sigma,KD) \quad src(\Delta) \models_d \phi[\eta] \quad \Delta \in MS(\Sigma)}{\Delta \models_d \phi[\eta]}$$

The proof can be found in App. C.3, page 216. Next we introduce the formula structure for the full preservation language; this language includes trace quantifiers $\textcircled{\exists}$ and $\textcircled{\forall}$.

Definition 5.4.2 (Preservation formulas) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA, a set of variables X suitable for Σ, a set R of role names suitable for TV, Σ, and a set $KD \subseteq KD(R, TV, \Sigma)$ of concept definitions. Then the formula structure $PF_{X,\Sigma,KD}$ of *preservation formulas over* X, Σ, KD is defined by

$$PF_{X,\Sigma,KD} := FOL((PF^{at}(X,\Sigma,KD), MS(\Sigma), \models_d, \mathcal{FV}), dom),$$

where *the set* $PF^{at}(X,\Sigma,KD)$ *of atomic preservation formulas over* X, Σ, KD, the satisfaction relation $\models_d$ and the function $\mathcal{FV}$ are determined as shown in Tab. 5.2. Moreover,

$$dom(\Delta, x:\tau, \eta) := \begin{cases} (src(\Delta), \{v \mid v \in \tau^{src(\Delta)}, ct \in CT^{min}_{src(\Delta)}(\Sigma, v) \\ \qquad\qquad (src(\Delta), ct) \overset{t_d}{\rightsquigarrow} v\}) & , \ \eta \in Env(X, src(\Delta)) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

for all $(\Delta, x:\tau, \eta) \in MS(\Sigma) \times X \times Env(X,\Sigma)$.
We refer to the components of $PF_{X,\Sigma,KD}$ by $PF(X,\Sigma,KD)$ (*preservation formulas over* X, Σ, KD), $\mathcal{FV}$ (free variables), and $\models_d$ (satisfaction relation). □

According to Tab. 5.2, atomic preservation formulas $PF^{at}(X,\Sigma,KD)$ comprise

(1) universally quantified trace formulas $\textcircled{\forall}\phi$, $\phi \in TF(X,\Sigma,KD)$,

(2) Σ-predicates and equality applied to Σ-terms and concept terms, which is expressed by $AT(KT_{X,\Sigma,KD} \uplus T_{X,\Sigma})$, and
(3) concept formulas $\phi \in KF(X, \Sigma, KD)$.

The set $PF(X, \Sigma, KD)$ of preservation formulas is determined by the FOPL-extension of the set of atomic preservation formulas so that general preservation formulas include negation, conjunction, and universal quantification.

Preservation formulas are evaluated w.r.t. a given migration sequence Δ. Atomic formulas over Σ-terms and concept terms as well as concept formulas are evaluated in the *source* state $src(\Delta)$ of Δ. In this way, pre-conditions can be specified. Apart from that, preservation formulas of the form $\forall o : \tau \bullet \phi \Rightarrow \psi$ can be used to select those objects (by the condition ϕ) that are subject to the preservation requirement ψ.

A preservation formula $\textcircled{\forall}\phi$ is satisfied, iff the trace formula ϕ holds for all inclusion-maximal trace tuples $\mathcal{TR}(\Delta)$ that have been extracted from Δ. This semantics is very similar to the semantics for path quantifiers in temporal logics. In the next lemma we state that the semantics of preservation formulas is well-defined. Notice that, in particular, the operation $KT_{X,\Sigma,KD} \uplus T_{X,\Sigma}$ is defined since concept terms and regular Σ-terms are disjoint by definition.

Again, we have to show that the so-defined formula structure is well-defined. The proof of the following lemma can be found in App. C.3, page 215.

Lemma 5.4.2 (Preservation formulas are well-defined) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA, a set of variables X suitable for Σ, a set R of role names suitable for TV, Σ, and a set $KD \subseteq KD(R, TV, \Sigma)$ of concept definitions. Then the formula structure $PF_{X,\Sigma,KD}$ is well-defined, i.e.:

(1) $KT_{X,\Sigma,KD} \uplus T_{X,\Sigma}$ is defined.
(2) For all $\Delta \in MS(\Sigma)$, $x : \tau \in X$, and $\eta \in Env(X, \Sigma)$ it is true that $dom(\Delta, x : \tau, \eta) = (\mathcal{A}, d)$ implies $\eta \in Env(X, \mathcal{A})$ and $d \subseteq \tau^{\mathcal{A}}$.

□

In the same way we use $\exists x \bullet \phi$ as a substitute for $\neg\forall x \bullet \neg\phi$, we use an existential trace quantifier $\textcircled{\exists}\phi$ for expressing $\neg\textcircled{\forall}\neg\phi$. This is also standard in temporal logics. $\textcircled{\exists}\phi$ holds if there is a trace tr such that ϕ holds for tr.

Definition 5.4.3 (Quantifier $\textcircled{\exists}$) Given a trace formula $\phi \in TF(X, \Sigma, KD)$, a migration sequence Δ over Σ, and a variable assignment η. Then $\Delta \models_d \textcircled{\exists}\phi[\eta]$ iff $\Delta \models_d \neg\textcircled{\forall}\neg\phi[\eta]$. □

Notice that the full integration of concept terms and concept formulas is an important feature of the preservation language. If only concept terms, concept formulas, and preservation constraints are used in a preservation formula ϕ, the so-defined formula fully abstracts from concrete implementations. The preservation requirement

$$
\begin{array}{l}
\forall w : \texttt{Website} \bullet \\
\quad \forall d, d' : \texttt{Dir} \bullet \\
\qquad \mathsf{WebSrc}(d, w) \wedge \mathsf{Contains}(d, d')[\mathit{CDirRec}] \Rightarrow \\
\qquad\quad \textcircled{\exists}\ d \mapsto \texttt{Dir} \wedge d' \mapsto \texttt{Dir} \wedge \texttt{pres}_{nf}(\mathsf{Contains}(d, d')[\mathit{CDirRec}, \mathit{CDirHtml}]) \wedge \\
\qquad\quad \textcircled{\exists}\ d \mapsto \texttt{Dir} \wedge d' \mapsto \texttt{Dir} \wedge \texttt{pres}_{nf}(\mathsf{Contains}(d, d')[\mathit{CDirRec}, \mathit{CDirRes}])
\end{array}
$$

is an example (cf. introduction to this chapter).

In App. B.3, page 193, we use selected preservation formulas to explain the semantics of preservation formulas in more detail. Also, our case study (Part IV) will contain more examples.

5.5 Summary

We have introduced syntax and semantics of a *preservation language*. First, we have defined a *preservation predicate pres* that relates source objects and target objects w.r.t. a given concept. If this concept is non-functional, this predicate holds if the concept *equally* holds for the source and target objects. In case of a functional concept, the concept must return equal values when applied to the source and target objects. This basic notion of preservation does not yet include object histories. However, we have shown that it works transitively. Iterated preservation results in global preservation. This is particularly important in our domain as migrations have to be executed periodically.

After that, we have supplemented this notion of preservation by object histories. For this purpose, we have introduced *object traces*. They are extracted from migration sequences Δ and represent parallel transformation paths for those objects that exist in the source state of Δ. Also, traces include system states and, hence, support expressive trace formulas. It could be proved that traces are circle-free; object histories are trees. This is a consequence of the invariants that hold in our dynamic system and keeps the language-semantics simple.

Traces are used to evaluate the semantics of *trace formulas*. Transformation and concept preservation constraints are atomic trace formulas. The former require objects to be transformed, the latter require that properties are preserved by transformations. By evaluating trace formulas w.r.t. traces that have been extracted from migration sequences, we fully integrate trace formulas into the dynamic environment that has been specified before. We have provided example trace formulas and derived their semantics in order to show this integration.

At that stage, the step towards a full preservation language was a small one. *Preservation formulas* include trace formulas that are quantified by trace quantifiers $\textcircled{\forall}$ and $\textcircled{\exists}$. These quantifiers determine whether a property must hold for *all* traces of a given migration sequence Δ or for *a single* trace only. Apart from that, preservation formulas include regular FOPL formulas over Σ terms, Σ formulas, concept terms, and concept formulas. These formulas can be understood as preconditions for migration processes or selectors for objects collections. They are connected to trace formulas by the usual logical operators. Overall, this results in an expressive language for preservation requirements, which can be evaluated in an automated way.

Part III

Improving Usability

Chapter 6

Implementing Migration Processes

Up to now, we describe migration algorithms by migration sequences (cf. Sect. 3.3.2). When dealing with complex migration tasks, however, the resulting lack of usability is unacceptable. Therefore, we introduce a functional programming language that can be used to implement complex migration algorithms conveniently. The language has a sound *operational semantics* based on ASMs and includes *basic state change operations* as regular functions. These operations *implement* the basic state changes of Sect. 3.3.2, i.e., they respect the corresponding pre- and post-conditions. By translating migration algorithms into equivalent migration sequences, we connect the functional language to our dynamic environment in a natural way. The agenda is as follows:

- In Sect. 6.1 we give an informal overview.
- Syntax and semantics of basic state change operations is introduced in Sect. 6.2. We will prove that the basic operations respect the pre- and post-conditions of the basic state changes introduced in Sect. 3.3.2.
- Syntax and semantics of the functional language is introduced in Sect. 6.3. In particular, this includes well-formedness and well-typedness rules for *expressions* of this language.
- We close with a short summary in Sect. 6.4.

6.1 Informal Overview

Our programming language must integrate object tracing; otherwise, preservation requirements cannot be evaluated automatically. As object traces are extracted from migration sequences, we define the semantics of our language in terms of migration sequences. This, in turn, requires that we implement the basic state changes appropriately.

Therefore, the functional language includes the following operations as regular functions:

(1) $\texttt{create}(f, \overline{t_i})$,
(2) $\texttt{transform}(t_{src} : \texttt{DObj} \mapsto (f, \overline{t_i}))$, and

(3) `delete`(t).

They implement the corresponding basic states change of Sect. 3.3.2 and particularly respect the relevant pre- and post conditions; this assures the desired system invariants.

Notice that `create`$(f, \overline{t_i})$ and `transform`$(t_{src} : \texttt{DObj} \mapsto (f, \overline{t_i}))$ automatically attach an ID to the result object. For this purpose, we introduce two additional administrative dynamic functions (apart from `existDObj` and `usedOIDs`). The ID that is to be used next is stored in `oidToUse`. Apart from that, `dep` stores object dependencies (in terms of $cont_{\texttt{DObj}}()$). That way we need not re-evaluate the object-valued content continuously when assuring the above-mentioned system invariants. The semantics of the basic state change operations is explained in the following example.

Example 6.1.1 (Effect of basic operations)

Recall that we have listed an example archive evolution in Tab. 3.1 on page 30. The table lists the values of the dynamic functions in the respective system states. In order to show how basic operations and basic state changes coincide, Tab. 6.1 shows the archive evolution that arises when executing the following basic *operations* that are equivalent to the basic state changes of Tab. 3.1:

(1)	$x_0 =$ `create`(`HTMLDoc`, "`start.html`", ...)	(State **0**)
(2)	$x_1 =$ `create`(`Dir`, "`calc05`", {}, {})	(State **1**)
(3)	$x_2 =$ `create`(`Dir`, "`overview`", {}, {})	(State **2**)
(4)	$x_3 =$ `create`(`Dir`, "`source`", $\{x_1, x_2\}$, $\{x_0\}$)	(State **3**)
(5)	$v_1 =$ `delete`(x_2)	(State **4**)
(6)	$v_2 =$ `delete`(x_3)	(State **4**)

We have assigned the created objects to variables x_0 to x_3 in order to emphasize that basic operations are *functions*. Apart from that, v_1 and v_2 carry the truth-values of the execution of the respective deletion operation. The system states in which the operations are executed are provided in parentheses.

□

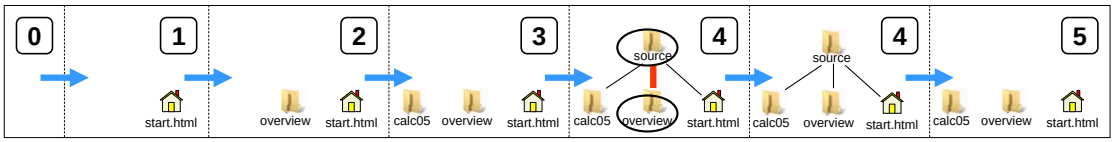
System states in Tab. 6.1 include the values for `oidToUse` and `dep`. In state zero, `oidToUse` stores the initial ID `initID`, which means that this ID will be attached to the next object on creation. Both `existDObj` and `dep` are empty. No objects and no object dependencies exist. While the four object creations take place, the value of `oidToUse` proceeds appropriately and `existDObj` is extended by the respective newly created object. Notice that the operation calls do not contain the object's ID. Attaching the ID (`oidToUse`) to the newly created objects is a built-in feature of the semantics. That way we assure the required system invariants.

All basic operations are *functions*. Object creation and object transformation return the created object; users have direct access to created objects. Object deletion reports success or failure by returning a truth value.

In state three, creating the "source" directory causes an update of `dep` because x_1 ("calc05"), x_2 ("overview"), and x_0 ("start.html") belong to the object-valued content of x_3. The tuple $(x_3, \{x_0, x_1, x_2\})$ in `dep` stores this dependency.

Next, a deletion operation is executed. It is blocked since x_2 ("overview") is part of the content of x_3 ("source"). Hence, v_1 is `False`. This dependency can be determined efficiently by a lookup in `dep`.

Table 6.1: *Example archive evolution using basic operations*



S	Value usedOIDs	Value oidToUse	Value existDObj	Value dep
0	{}	initID	{}	{}
1	{initID}	nextID(initID)	{HTMLDoc(initID, "start.html", ...)}	{}
2	{initID, nextID(initID)}	nextID^2(initID)	{HTMLDoc(initID, "start.html", ...), Dir(nextID(initID), "calc05", {}, {})}	{}
3	{initID, nextID(initID), nextID^2(initID)}	nextID^3(initID)	{HTMLDoc(initID, "start.html", ...), Dir(nextID(initID), "calc05", {}, {}), Dir(nextID^2(initID), "overview", {}, {}.)}	{}
4	{initID, nextID(initID), nextID^2(initID), nextID^3(initID)}	nextID^4(initID)	{HTMLDoc(initID, "start.html", ...), Dir(nextID(initID), "calc05", {}, {}), Dir(nextID^2(initID), "overview", {}, {}), Dir(nextID^3(initID), "source", $\{x_1, x_2\}$, $\{x_0\}$)}	$\{(x_3, \{x_0, x_1, x_2\})\}$
4	{initID, nextID(initID), nextID^2(initID), nextID^3(initID)}	nextID^4(initID)	{HTMLDoc(initID, "start.html", ...), Dir(nextID(initID), "calc05", {}, {}), Dir(nextID^2(initID), "overview", {}, {}), Dir(nextID^3(initID), "source", $\{x_1, x_2\}$, $\{x_0\}$)}	$\{(x_3, \{x_0, x_1, x_2\})\}$
5	{initID, nextID(initID), nextID^2(initID), nextID^3(initID)}	nextID^4(initID)	{HTMLDoc(initID, "start.html", ...), Dir(nextID(initID), "calc05", {}, {}), Dir(nextID^2(initID), "overview", {}, {})}	{}

The example trace in Tab. 6.1 suggests that the basic state change operations satisfy the desired pre- and post-conditions. In particular, $id <_{id}$ `oidToUse` holds for all $id \in$ `usedOIDs` such that indeed fresh IDs are used for newly created objects. Also, the effect directly corresponds to the effect of the respective basic state changes in Tab. 3.1. Altogether, the semantics of the algorithm in Tab. 6.1 yields a migration sequence of length five.

It is, however, impracticable to provide a basic operation for each single migration step as migrations get complex. First, these migration sequences are not re-usable. Second, it is hardly feasible to provide basic operations explicitly for thousands of digital objects. Therefore, we integrate these operations into a full functional programming language. In Fig. 6.1 we show two sample function definitions that are related to our case study (Part IV); by copying a directory structure and contained html files recursively, they partly *assure BWeb* conformity of transformed websites.

There,

$$\texttt{migDirHTML} : \texttt{Dir} \to \texttt{Website} \to \texttt{Dir} \text{ and}$$
$$\texttt{migHTMLDoc} : \texttt{HTMLDoc} \to \texttt{Website} \to \texttt{HTMLDoc}$$

are called *function declarations* as they specify name and typing of the respective function. The parts $\texttt{migDirHTML}(d, w) = \texttt{let}...$ and $\texttt{migHTMLDoc}(d, w) = \texttt{if}...$ are called *function body*.

The functions in Fig. 6.1 exhibit most of the features that are supported by the functional language. This includes

(1) set comprehension ($\{\texttt{migDirHTML}(d', w) \mid d' \leftarrow \texttt{subDirs}(d), \texttt{True}\}$),

(2) a `let` construct,

Function `migDirHTML`**:**

```
migDirHTML : Dir → Website → Dir
migDirHTML(d, w) =
   let  sDirs  =  {migDirHTML(d', w) | d' ← subDirs(d), True}
        sDocs  =  {migHTMLDoc(d', w) | d' ← subDocs(d),
                                       and(not(d' = home(w)), inst_HTMLDoc(d'))}
   in   transform(d ↦ (Dir, name(d), sDirs, sDocs))
```

Function `migHTMLDoc`**:**

```
migHTMLDoc : HTMLDoc → Website → HTMLDoc
migHTMLDoc(d, w) =
  if not(d = home(w))
     then  transform(d ↦ (HTMLDoc, name(d), content(d)))
     else  transform(d ↦ (HTMLDoc, "index.html", content(d)))
```

Figure 6.1: Example algorithm using the functional language

(3) an `if...then...else...` construct,

(4) integration of basic operations (`transform`$(d \mapsto (\texttt{Dir}, \texttt{name}(d), sDirs, sDocs))$),

(5) integration of components of existing Σ-models, which includes

- the instance operator ($\texttt{inst}_{\texttt{HTMLDoc}}(d')$),
- equality ($d' = \texttt{home}(w)$) as well as other predicates, and
- functions ($\texttt{home}(w)$), and

(6) recursion (in the body of `migDirHTML`).

The semantics of `let`, `if`, and set comprehension is largely standard. However, the semantics additionally generates an equivalent migration sequence.

Example 6.1.2 (Effect of migration algorithms)

Let w denote a website that contains the file and directory structure of state four in Tab. 6.1. Hence,

$\texttt{srcDir}(w) = x_3$ ("source")
$\texttt{home}(w) = x_0$ ("start.html")
$x_3 = \texttt{Dir}(\texttt{nextID}^3(\texttt{initID}), \text{“}\texttt{source}\text{“}, \{x_1, x_2\}, \{x_0\})$

using abbreviations of Example 6.1.1. Notice that w corresponds to a part of the example website "Calculation" in Fig. 2.2 on page 17 such that this example directly carries over to the running example.

The function call $\texttt{migDirHTML}(x_3, w)$ transforms the file and directory structure of w and results in the following migration sequence

$$\langle\ \mathcal{A}_0,\ \mathsf{trans}(x_1 \mapsto x_1'),\ \mathcal{A}_1,\ \mathsf{trans}(x_2 \mapsto x_2'),\ \mathcal{A}_2,\ \mathsf{trans}(x_0 \mapsto x_0'),\ \mathcal{A}_3,\ \mathsf{trans}(x_3 \mapsto x_3'),\ \mathcal{A}_4\rangle,$$

where we have abbreviated

$$\begin{aligned}
x_0' &= \texttt{HTMLDoc}(\texttt{nextID}^6(\texttt{initID}), \text{“}\texttt{index.html}\text{“}, ...),\\
x_1' &= \texttt{Dir}(\texttt{nextID}^4(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\}),\\
x_2' &= \texttt{Dir}(\texttt{nextID}^5(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{\}),\\
x_3' &= \texttt{Dir}(\texttt{nextID}^7(\texttt{initID}), \text{“}\texttt{source}\text{“}, \{x_1', x_2'\}, \{x_0'\}).
\end{aligned}$$

□

Notice that the execution order is important in this example. The result is a migration *sequence* and all object-valued content of an object must exist before this object can be created. In particular, x_3' is created last because it contains x_0', x_1', and x_2'. This execution order is indeed assured in the body of `migDirHTML`. First, all sub-directories of x_3 are transformed. The result is stored in $sDirs$. Then x_0 is transformed, which is stored in $sDocs$. Finally, d is transformed while setting $sDirs$ and $sDocs$ as `subDirs` and `subDocs` attribute of the result directory.

In the following sections we introduce the necessary formalisms. At the end of this chapter we can conveniently specify migration algorithms and have a sound operational semantics based on migration sequences that fully integrates the functional programming language into our dynamic environment.

6.2 Basic State Change Operations

Here we provide an operational semantics for the basic operations that exactly respects the required pre- and post-conditions of Sect. 3.3.2. In order to achieve this, we define all operations in terms of ASM programs. That way, the *effect of basic operations* is determined by a new system state that arises from executing an appropriate ASM program. This has the following advantages:

- We can prove that the basic operations adhere to the pre-and post-conditions and, thus, conclude that the invariants and properties of Sect. 3.3.2 hold.
- ASMs are "operational by nature", which yields executable specifications.
- Users can extend the basic operations if desired. If this extension is a *refinement*, the properties of the dynamic system are preserved.
- The ASM theory is well-established [Gur00, GS97, Rei03b, Rei03a]. It provides theories and tools for reasoning about programs and program refinements ([BS03, Ton98]) and has been integrated with abstract datatypes as well ([Zam98, Zam97]).

Before we provide the semantics of the basic operations, we recall some preliminaries about ASMs ([BS03]). ASMs are used to model dynamic systems and view an *abstract state* as a kind of memory that maps *locations* to values.

Definition 6.2.1 (Location) Given a signature $\Sigma := \Sigma_s \cup \Sigma_d$ and a Σ-state $\mathcal{A}$. A *location of* $\mathcal{A}$ is a pair $loc := (f \bullet \overline{v_i})$, where $f : \tau_1 \times ... \times \tau_n \to \tau$ is an n-ary function symbol of $\mathcal{F}_s \cup \mathcal{F}_d$ and $v_i \in \tau_i^{\mathcal{A}}$. The value $f^{\mathcal{A}}(\overline{v_i^{\mathcal{A}}})$ is called the *content* of the location loc. Moreover, v_i are the *elements* of loc. We call loc *dynamic*, if $f \in \mathcal{F}_d$ and *static*, otherwise. □

In this respect, we can view states $\mathcal{A}$ as functions that map the locations of $\mathcal{A}$ to values of the corresponding type domain in $\mathcal{A}$. We will denote the *content of the location loc in* $\mathcal{A}$ by $\mathcal{A}(loc)$. The basic idea behind abstract state machines is to let the dynamic locations evolve. For this purpose, the semantics, which we introduce later on, produces appropriate *location updates*.

Definition 6.2.2 (Update) Given a signature $\Sigma := \Sigma_s \cup \Sigma_d$ and a Σ-state $\mathcal{A}$. An *update* for $\mathcal{A}$ is a mapping $((f \bullet \overline{v_i}) \mapsto v)$, where $(f \bullet \overline{v_i})$ is a dynamic location of $\mathcal{A}$, $f : \tau_1 \times ... \times \tau_n \to \tau$, and $v \in \tau^{\mathcal{A}}$. An *update set* is a set of updates. □

Table 6.2: *Syntax and semantics of ASM transition rules*

Basic ASM rules:			
(1)	Skip rule:	`skip`	(Do nothing)
(2)	Update rule:	$f(\overline{s_i}) := t$	(Update f at $\overline{s_i}$ to t)
(3)	Block rule:	`par` $P_1, ..., P_n$`end par`	(Execute $P_1, ..., P_n$ in parallel)

Semantics:

(1) $$\frac{}{[\![\texttt{skip}]\!]^{\mathcal{A}}_{\eta} = \emptyset}$$

(2) $$\frac{}{[\![f(\overline{s_i}) := t]\!]^{\mathcal{A}}_{\eta} = \{((f \bullet \overline{\mathcal{V}^{\mathcal{A}}[\![s_i]\!]\eta}) \mapsto \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta)\}}$$

(3) $$\frac{[\![P_1]\!]^{\mathcal{A}}_{\eta} = U_1 \quad ... \quad [\![P_n]\!]^{\mathcal{A}}_{\eta} = U_n}{[\![\texttt{par } P_1, ..., P_n \texttt{ end par}]\!]^{\mathcal{A}}_{\eta} = \bigcup_{1 \le i \le n} U_i}$$

(4.1) $$\frac{[\![P]\!]^{\mathcal{A}}_{\eta} = U \qquad \mathcal{A} \models \phi[\eta]}{[\![\texttt{if } \phi \texttt{ then } P \texttt{ else } Q \texttt{ end if}]\!]^{\mathcal{A}}_{\eta} = U}$$

(4.2) $$\frac{[\![Q]\!]^{\mathcal{A}}_{\eta} = V \qquad \mathcal{A} \not\models \phi[\eta]}{[\![\texttt{if } \phi \texttt{ then } P \texttt{ else } Q \texttt{ end if}]\!]^{\mathcal{A}}_{\eta} = V}$$

Two or more updates clash if they update the same location with different values. Hence, we need a notion of consistency for update sets.

Definition 6.2.3 (Consistent update sets) An update set U is called *consistent* (denoted by $\texttt{Cons}(U)$), if for any location loc and elements v, v' the following holds: $(loc \mapsto v) \in U$ and $(loc \mapsto v') \in U$ implies $v = v'$. □

Consistent update sets can be *fired* at a given state $\mathcal{A}$. This updates the dynamic locations of $\mathcal{A}$ and leaves the static parts of $\mathcal{A}$ unchanged.

Definition 6.2.4 (Firing of updates) Given a signature Σ, a Σ-state $\mathcal{A}$, and a consistent update set U. The result of *firing* U *at* $\mathcal{A}$ is a new state $\mathcal{A} + U$ such that for every location loc of $\mathcal{A}$

$$(\mathcal{A} + U)(loc) = \begin{cases} v, & (loc \mapsto v) \in U \\ \mathcal{A}(l), & \text{otherwise} \end{cases} .$$

The state $\mathcal{A} + U$ is called the *sequel of* $\mathcal{A}$ *w.r.t.* U. □

Update sets being consistent assure that $\mathcal{A} + U$ is uniquely determined. With these preliminaries we are ready to introduce basic transition rules for abstract state machines.

Definition 6.2.5 (ASM transition rules) Given a signature $\Sigma := \Sigma_s \cup \Sigma_d$ and a dynamic function symbol $f \in \mathcal{F}_d$. Then syntax and semantics of *ASM transition rules* are defined inductively as shown in Tab. 6.2. □

The semantics of an ASM program is the update set it yields. Although Tab. 6.2 covers only parts of the ASM syntax and semantics provided in [BS03], it is sufficient for defining our basic state change operations.

Table 6.3: *Syntax of basic operations $\mathcal{BO}p(X,\Sigma)$ over Σ, X*

Syntax:		
(1)	**Object creation:** $\dfrac{f : \texttt{OID} \times \overline{\tau_i} \to \tau \in \mathcal{C}_\tau \quad t_1 \in T_{\tau_1}(X,\Sigma), ..., t_n \in T_{\tau_n}(X,\Sigma) \qquad \tau{<}\texttt{DObj}}{\texttt{create}(f, \overline{t_i}) \in \mathcal{BO}p_\tau(X,\Sigma)}$	$\mathcal{FV}(\texttt{create}(f, \overline{t_i})) = \bigcup_i \mathcal{FV}(t_i)$
(2)	**Object transformation:** $\dfrac{t_{src} \in T_{\texttt{DObj}}(X,\Sigma) \quad f : \texttt{OID} \times \overline{\tau_i} \to \tau \in \mathcal{C}_\tau \quad t_1 \in T_{\tau_1}(X,\Sigma), ..., t_n \in T_{\tau_n}(X,\Sigma) \qquad \tau{<}\texttt{DObj}}{\texttt{transform}(t_{src} \mapsto (f, \overline{t_i})) \in \mathcal{BO}p_\tau(X,\Sigma)}$	$\mathcal{FV}(\texttt{transform}(t_{src} \mapsto (f, \overline{t_i}))) = \mathcal{FV}(t_{src}) \cup \bigcup_i \mathcal{FV}(t_i)$
(3)	**Object deletion:** $\dfrac{t \in T_{\texttt{DObj}}(X,\Sigma)}{\texttt{delete}(t) \in \mathcal{BO}p_{\texttt{Bool}}(X,\Sigma)}$	$\mathcal{FV}(\texttt{delete}(t)) = \mathcal{FV}(t)$
(4)	**Subtyping:** $\dfrac{op \in \mathcal{BO}p_{\tau'}(X,\Sigma) \quad \tau'{<}\tau}{op \in \mathcal{BO}p_\tau(X,\Sigma)}$	

In the introduction to this chapter we mentioned that we use additional dynamic functions for the specification of the basic operations. In particular, we store object dependencies in `dep` and the next free object ID in `oidToUse`. Before we introduce the syntax of basic operations, we, thus define suitable signatures for their evaluation.

Definition 6.2.6 (Signature suitable for basic operations) A signature Σ is *suitable for evaluating basic operations*, iff

$$\{\texttt{oidToUse} : \texttt{OID}, \texttt{dep} : \texttt{DObj} \to \texttt{Set[DObj]}\} \cup \mathcal{F}_d^{DA} \subseteq \mathcal{F}_{dp(\Sigma)}.$$

□

Signatures that are suitable for evaluating basic operations at least contain the dynamic functions of the basic DA as well as `oidToUse` and `dep`. Now, we are ready to define the syntax of basic operations; some examples are listed in App. B.4, page 193.

Definition 6.2.7 (Basic operations) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA such that Σ is suitable for evaluating basic operations. Given, furthermore, a suitable set of variables X for Σ. Then the set $\mathcal{BO}p(X,\Sigma)$ of *basic state change operations over Σ and X* and the free variables $\mathcal{FV}(op)$, $op \in \mathcal{BO}p(X,\Sigma)$, are defined as shown in Tab. 6.3. □

The effect of basic operations is determined by firing the corresponding ASM rule in Tab. 6.4. Object creation is implemented by *r_create*, which updates `oidToUse` and extends `existDObj`, `usedOIDs`, and `dep`. Notice that `dep` is set to t_{sub}, which is a parameter of the rule application. It contains the object-valued content of the parameter terms $\overline{t_i}$ as we will see shortly; `dep` stores object dependencies according to $cont_{\texttt{DObj}}()$.

The rule *r_transform* implements object transformation and works similar to *r_create*. However, it explicitly requires t_{src} to exist. As our basic DA does not administrate object histories, *r_transform* is relatively simple. As all other rules, it, however, can be extended if desired; we merely require extensions to be *refinements*.

Table 6.4: *ASM rule declarations for the basic state change operations*

Rule declarations *r_create* **and** *r_delete*:

r_create$(f, \overline{t_i}, t_{sub})$ ::=
if $\forall x$: DObj • $x \in t_{sub} \Rightarrow x \in$ existDObj
then
  par
    oidToUse := nextID(oidToUse)
    existDObj :=
     $\{f(\text{oidToUse}, \overline{t_i})\}_s \cup$ existDObj
    usedOIDs :=
     $\{\text{oidToUse}\}_s \cup$ usedOIDs
    dep$(f(\text{oidToUse}, \overline{t_i})) := t_{sub}$
  end par
else
  skip
end if

r_delete(t) ::=
if $t \in$ existDObj$\wedge$
   $\forall x$: DObj • $x \in$ existDObj $\Rightarrow t \notin$ dep(x)
then
  par
    existDObj := existDObj$\backslash\{t\}_s$
    dep$(t) := \{\}$
  end par
else
  skip
end if

Rule declaration *r_transform*:

r_transform$(t_{src}, f, \overline{t_i}, t_{sub})$ ::=
if $t_{src} \in$ existDObj $\wedge \forall x$: DObj • $x \in t_{sub} \Rightarrow x \in$ existDObj then
  par
    oidToUse := nextOID(oidToUse)
    existDObj := $\{f(\text{oidToUse}, \overline{t_i})\}_s \cup$ existDObj
    usedOIDs := $\{\text{oidToUse}\}_s \cup$ usedOIDs
    dep$(f(\text{oidToUse}, \overline{t_i})) := t_{sub}$
  end par
else
  skip
end if

Finally, deletion is implemented by *r_delete*. This rule removes the corresponding object from existDObj and deletes all dependencies. Notice that all operations have certain pre-conditions. If they are not satisfied, the operations have no effect on the system state.

In the following we define the effect of basic operations formally in terms of the ASM rules of Tab. 6.4.

Definition 6.2.8 (Effect of basic operations) Given a specification $Spec := (\Sigma, Sen)$ that is a permissible extension of the basic DA such that Σ is suitable for evaluating basic operations, a Σ-state $\mathcal{A}$, a migration sequence Δ over Σ, a suitable set of variables X for Σ, and a basic operation $op \in \mathcal{BO}p(X, \Sigma)$. Then the *effect of op in* $\mathcal{A}$ *under* η (denoted by $(\mathcal{A}, \Delta, \eta, op) \overset{op}{\rightsquigarrow} (\mathcal{A}', \Delta', v)$) is defined by the rules of Tab. 6.5. □

Examples can be found in App. B.4, page 194. Basic operations result in a system state, a migration sequence, and a value. With object deletion, the truth value directly indicates the success of the operation. In the other two cases success can be verified by checking whether the returned value is in existDObj or not. This allows to adjust the control flow of algorithms according to an operation's success.

Table 6.5: *Effect of basic operations*

Semantics:

(1) **Object creation 1:**

$$\frac{\begin{array}{ll}\texttt{create}(f,\overline{t_i}) \in \mathcal{BO}p_\tau(X,\Sigma) & [\![r\_create(f,\overline{t_i},T^{\leq}_{\texttt{DObj}}(\emptyset,ct)\backslash\{ct\})]\!]^{\mathcal{A}}_{\eta} = U \\ ct \in CT^{min}_{\mathcal{A}}(\Sigma,v) & (\mathcal{A},\eta,f(\texttt{oidToUse},\overline{t_i})) \overset{t}{\rightsquigarrow} v, U = \emptyset\end{array}}{(\mathcal{A},\Delta,\eta,\texttt{create}(f,\overline{t_i})) \overset{op}{\rightsquigarrow} (\mathcal{A},\Delta,v)}$$

(2) **Object creation 2:**

$$\frac{\begin{array}{ll}\texttt{create}(f,\overline{t_i}) \in \mathcal{BO}p_\tau(X,\Sigma) & [\![r\_create(f,\overline{t_i},T^{\leq}_{\texttt{DObj}}(\emptyset,ct)\backslash\{ct\})]\!]^{\mathcal{A}}_{\eta} = U \\ ct \in CT^{min}_{\mathcal{A}}(\Sigma,v) & (\mathcal{A},\eta,f(\texttt{oidToUse},\overline{t_i})) \overset{t}{\rightsquigarrow} v, U \neq \emptyset\end{array}}{(\mathcal{A},\Delta,\eta,\texttt{create}(f,\overline{t_i})) \overset{op}{\rightsquigarrow} (\mathcal{A}+U,\Delta;\langle\mathcal{A},\mathsf{cre}(ct),\mathcal{A}+U\rangle,v)}$$

(3) **Object transformation 1:**

$$\frac{\begin{array}{ll}\texttt{transform}(t_{src}\mapsto(f,\overline{t_i})) & [\![r\_transform(t_{src},f,\overline{t_i},T^{\leq}_{\texttt{DObj}}(\emptyset,ct_{trg})\backslash\{ct_{trg}\})]\!]^{\mathcal{A}}_{\eta} = U \\ \in \mathcal{BO}p_\tau(X,\Sigma) & (\mathcal{A},\eta,f(\texttt{oidToUse},\overline{t_i})) \overset{t}{\rightsquigarrow} v,\ U = \emptyset \\ ct_{trg} \in CT^{min}_{\mathcal{A}}(\Sigma,v) & \end{array}}{(\mathcal{A},\Delta,\eta,\texttt{transform}(t_{src}\mapsto(f,\overline{t_i}))) \overset{op}{\rightsquigarrow} (\mathcal{A},\Delta,v)}$$

(4) **Object transformation 2:**

$$\frac{\begin{array}{ll}\texttt{transform}(t_{src}\mapsto(f,\overline{t_i})) & [\![r\_transform(t_{src},f,\overline{t_i},T^{\leq}_{\texttt{DObj}}(\emptyset,ct_{trg})\backslash\{ct_{trg}\})]\!]^{\mathcal{A}}_{\eta} = U \\ \in \mathcal{BO}p_\tau(X,\Sigma) & (\mathcal{A},\eta,f(\texttt{oidToUse},\overline{t_i})) \overset{t}{\rightsquigarrow} v,\ U \neq \emptyset \\ ct_{src} \in CT^{min}_{\mathcal{A}}(\Sigma,\mathcal{V}^{\mathcal{A}}[\![t_{src}]\!]\eta) & \\ ct_{trg} \in CT^{min}_{\mathcal{A}}(\Sigma,v) & \end{array}}{(\mathcal{A},\Delta,\eta,\texttt{transform}(t_{src}\mapsto(f,\overline{t_i}))) \overset{op}{\rightsquigarrow} (\mathcal{A}+U,\Delta;\langle\mathcal{A},\mathsf{trans}(ct_{src}\mapsto ct_{trg}),\mathcal{A}+U\rangle,v)}$$

(5) **Object deletion 1:**

$$\frac{\texttt{delete}(t) \in \mathcal{BO}p_{\texttt{Bool}}(X,\Sigma) \qquad [\![r\_delete(t)]\!]^{\mathcal{A}}_{\eta} = U,\ U = \emptyset}{(\mathcal{A},\Delta,\eta,\texttt{delete}(t)) \overset{op}{\rightsquigarrow} (\mathcal{A},\Delta,\texttt{False}^{\mathcal{A}})}$$

Object deletion 2:

$$\frac{\begin{array}{ll}\texttt{delete}(t) \in \mathcal{BO}p_{\texttt{Bool}}(X,\Sigma) & [\![r\_delete(t)]\!]^{\mathcal{A}}_{\eta} = U,\ U \neq \emptyset \\ ct \in CT^{min}_{\mathcal{A}}(\Sigma,\mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) & \end{array}}{(\mathcal{A},\Delta,\eta,\texttt{delete}(t)) \overset{op}{\rightsquigarrow} (\mathcal{A}+U,\Delta;\langle\mathcal{A},\mathsf{del}(ct),\mathcal{A}+U\rangle,\texttt{True}^{\mathcal{A}})}$$

The result state is determined by firing an appropriate ASM rule as defined in Tab. 6.4. Notice the value of the parameter t_{sub} in rules **Object creation** and **Object transformation**. It equals the set of true subterms of ct/ct_{trg} of type `DObj` or below. Since ct and ct_{trg} are minimal constructor terms, their subterms determine the object-valued content for the object that is to be created. Since these terms have to be generated anyway at this point, we could require minimal constructor terms as arguments for the basic state changes (cf. Defn. 3.3.7, page 48) without causing additional system overhead (thus, runtime costs). Continuative explanations on Defn. 6.2.8 can be found in App. B.4, page 195.

Lemma 6.2.1 (Effect of basic operations is well-defined) The effect of basic operations is well-defined, i.e., given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$ such that Σ is suitable

for evaluating basic operations, a set X of variables suitable for Σ, a Σ-algebra $\mathcal{A}$, a variable assignment η, a migration sequence Δ, and a basic operation $op \in \mathcal{BO}p_\tau(X, \Sigma)$, there is exactly one $\mathcal{A}'$, a migration sequence Δ', and exactly one element $v \in \tau^{\mathcal{A}} \cup \{\bot\}$ such that $(\mathcal{A}, \Delta, \eta, op) \overset{op}{\rightsquigarrow} (\mathcal{A}', \Delta', v)$. □

The proof can be found in App. C.1, page 217, and goes by straightforward induction on the structure of *op*. Basic operations produce a unique migration sequence Δ' only if they fail (then $\Delta' = \Delta$). Otherwise, the basic state change that is appended to Δ contains a minimal constructor term, which is not uniquely determined *syntactically*. We will prove shortly that the basic operations generate appropriate migration sequences on the semantic level and adhere to the pre- and post-conditions of the basic state changes. (cf. Defn. 3.3.7). This is prepared by the following lemma.

Lemma 6.2.2 (Basic operations yield consistent updates) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA such that Σ is suitable for evaluating basic operations, a set of variables X suitable for Σ, and a basic operation $op \in \mathcal{BO}p_{\texttt{DObj}}(X, \Sigma)$. Then *op* yields a consistent update, i.e., $[\![op]\!]^{\mathcal{A}}_{\eta}$ is consistent for all Σ-states $\mathcal{A} \in A(\Sigma)$ and all variable assignments $\eta \in Env(X, \mathcal{A})$. □

The proof can be found in App. C.1, page 217. It generates the resulting update sets using the calculus in Tab. 6.2. According to the following theorem, the semantics of basic operations produces valid state changes w.r.t. $\overset{btr}{\rightsquigarrow}$.

Theorem 6.2.1 (Basic operations yield valid basic state changes) Given a permissible extension $Spec := (\Sigma, Sen)$ of the basic DA such that Σ is suitable for evaluating basic operations. Then the basic state change operations generate valid basic state changes, i.e.,

(1) $(\mathcal{A}, \Delta, \eta, \texttt{create}(f, \overline{t_i})) \overset{op}{\rightsquigarrow} (\mathcal{A}', \Delta; \langle \mathcal{A}, \mathsf{cre}(ct), \mathcal{A}' \rangle, v)$ implies $(\mathcal{A}, \mathsf{cre}(ct)) \overset{btr}{\rightsquigarrow} \mathcal{A}'$.

(2) $(\mathcal{A}, \Delta, \eta, \texttt{transform}(t_{src} \mapsto (f, \overline{t_i}))) \overset{op}{\rightsquigarrow} (\mathcal{A}', \Delta; \langle \mathcal{A}, \mathsf{trans}(ct_{src} \mapsto ct_{trg}), \mathcal{A}' \rangle, v)$ implies $(\mathcal{A}, \mathsf{trans}(ct_{src} \mapsto ct_{trg})) \overset{btr}{\rightsquigarrow} \mathcal{A}'$.

(3) $(\mathcal{A}, \Delta, \eta, \texttt{delete}(t)) \overset{op}{\rightsquigarrow} (\mathcal{A}', \Delta; \langle \mathcal{A}, \mathsf{del}(ct), \mathcal{A}' \rangle, v)$ implies $(\mathcal{A}, \mathsf{del}(ct)) \overset{btr}{\rightsquigarrow} \mathcal{A}'$.

□

The proof in App. C.1, page 218, is straightforward by showing validity of the required pre- and post-conditions of Defn. 3.3.7. Also, it relies on validity of Lemma 6.2.2.

Due to Thm. 6.2.1 all system invariants of Lemma 3.3.5 hold. Notice that the correspondence between basic operations and basic state changes can be proved only for *permissible* extensions of the basic DA (cf. Defn. 3.3.6). Only the constructor property (see Defn. 3.3.2, page 43) holding facilitates to trace object dependencies in `dep` using *subterms* of the object that is to be created. This concludes the formal parts concerning basic operations. In the next section we introduce migration algorithms and our programming language.

6.3 Migration Algorithms

In the following, we will use the example function definitions of Fig. 6.1 on page 90 to explain different aspects of the introduced programming language. As an important

feature, our language *extends* a given signature Σ. This results in smooth integration of migration algorithms into specifications for digital archives; we can re-use Σ-models and need not explicitly re-implement functions and predicates. For this purpose, our language uses predicates of the signature as well as equality $=$ as boolean-valued functions. The first step, thus, is to translate these parts appropriately. This translation yields the *generated signature* $gen(\Sigma)$ for Σ.

Definition 6.3.1 (Generated signature) Given a permissible extension $Spec(\Sigma, \mathcal{S}en)$ of the basic DA where $sp(\Sigma) := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$. Then the *generated signature* $gen(\Sigma)$ for Σ is defined by

$$sp(gen(\Sigma)) := (\mathcal{T}, <, \emptyset, \mathcal{C}, \mathcal{F} \cup \{p : \overline{\tau_i} \to \texttt{Bool} \mid p : \overline{\tau_i} \in \mathcal{P}\} \cup \{=: \texttt{Top} \times \texttt{Top} \to \texttt{Bool}\}).$$

and $dp(gen(\Sigma)) = dp(\Sigma)$. Also, we extend Σ-algebras $\mathcal{A}$ to $gen(\Sigma)$-algebras $gen(\Sigma, \mathcal{A})$ by the following interpretations (denoting $gen(\Sigma, \mathcal{A})$ by $\mathcal{A}'$).

$$p^{\mathcal{A}'}(v_1, ..., v_n) := \begin{cases} \texttt{True}^{\mathcal{A}}, & v_1 \in \tau_1^{\mathcal{A}}, ..., v_1 \in \tau_n^{\mathcal{A}}, (v_1, ..., v_n) \in p^{\mathcal{A}} \\ \texttt{False}^{\mathcal{A}}, & v_1 \in \tau_1^{\mathcal{A}}, ..., v_1 \in \tau_n^{\mathcal{A}}, (v_1, .., v_n) \notin p \\ \bot, & \text{otherwise} \end{cases} \quad \text{for } p_{\overline{\tau_i}} \in \mathcal{P}$$

$$=^{\mathcal{A}'}(v, v') := \begin{cases} \texttt{True}^{\mathcal{A}}, & v, v' \in \texttt{Top}^{\mathcal{A}} \wedge v = v' \\ \texttt{False}^{\mathcal{A}}, & v, v' \in \texttt{Top}^{\mathcal{A}} \wedge v \neq v' \\ \bot, & \text{otherwise} \end{cases}$$

□

The generated signature always contains equality $=: \texttt{Top} \times \texttt{Top} \to \texttt{Bool}$ and Σ-predicates as Boolean-valued functions. This construction is well-defined because (1) Σ contains types `Bool` and `Top` ($Spec$ is a permissible extension of $Spec^{DA}$) and (2) we require function and predicate names to be disjoint (cf. Defn. 3.2.1).

Σ-algebras are extended to $gen(\Sigma, \mathcal{A})$-algebras by introducing *strict* interpretations for the new Boolean-valued functions; equality $=$ and predicates return $\bot$ if one of the arguments evaluates to $\bot$. According to Defn. 3.2.3 we have to show that the new functions respect overloading.

Lemma 6.3.1 ($gen(\Sigma, \mathcal{A})$ is a well-defined algebra) Given a permissible extension $Spec(\Sigma, \mathcal{S}en)$ of the basic DA and an algebra $\mathcal{A} \in A(\Sigma)$. Then $gen(\Sigma, \mathcal{A})$ is a well-defined algebra according to Defn. 3.2.3. □

The proof in App. C.4, page 222, exploits that predicates of Σ respect overloading. This property is preserved when translation $\Sigma, \mathcal{A}$ to $gen(\Sigma)$ and $gen(\Sigma, \mathcal{A})$, respectively. The value relation $\overset{t}{\rightsquigarrow}$ for Σ-terms naturally extends to $gen(\Sigma)$-terms. Since $gen(\Sigma)$ contains a greatest type `Top` (Σ extends Σ^{DA}), derived term values are unique (cf. Lemma 3.2.3, page 37).

When developing complex migration algorithms, we are likely to declare new functions (like `migDirHTML` and `migHTMLDoc` in Fig. 6.1) that are not included in Σ^{DA}. They are, however, typed over $\mathcal{T}$. This is captured by the notion of *function declarations* that are *suitable for* Σ.

Definition 6.3.2 (Function declarations) Given a signature $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$. Then a set of *function declarations* $\mathcal{F}_{decl}$ *is suitable for* Σ, iff $\mathcal{F}_{decl}$ contains only non-overloaded function symbols, all elements $f : \overline{\tau_i} \to \tau \in \mathcal{F}_{decl}$ are typed over $\mathcal{T}$, and all function names in $N_{\mathcal{F}_{decl}} := \{f \mid f : \overline{\tau_i} \to \tau \in \mathcal{F}_{decl}\}$ differ from all symbol names of Σ, $=$, and `cast`, i.e., $N_{\mathcal{F}_{decl}} \cap (N_{\mathcal{F}} \cup N_{\mathcal{P}} \cup \{=, \texttt{cast}\}) = \emptyset$. □

As explained before, this definition particularly requires fresh symbols in $\mathcal{F}_{decl}$ that are typed over $\mathcal{T}$. Notice that we require these symbols to be distinct from the typecast operator `cast`, which will be part of our expression syntax. Also, these function symbols may *not* be overloaded. This keeps our semantics simple and expressiveness does not suffer anyway.

In the following definition we introduce *expressions* and *function definitions*.

Definition 6.3.3 (Expressions, function definitions) Given a permissible extension $Spec(\Sigma, \mathcal{S}en)$ of the basic DA such that Σ is suitable for evaluating basic operations and a set $\mathcal{F}_{decl}$ of function declarations suitable for Σ. Then

(1) the set $E_\tau(X, \Sigma, \mathcal{F}_{decl})$ of *expressions of type* τ *over* X, Σ, *and* $\mathcal{F}_{decl}$, and

(2) the set of free variables $\mathcal{FV}(e)$ for $e \in E(X, \Sigma, \mathcal{F}_{decl})$

are defined as shown in the upper part of Tab. 6.6. Moreover,

(3) the set $FD_\tau(X, \Sigma, \mathcal{F}_{decl})$ of *function definitions of type* τ *over* Σ, $\mathcal{F}_{decl}$, *and* X

is defined as shown in the lower part of Tab. 6.6.
A set $FD \subseteq FD(X, \Sigma, \mathcal{F}_{decl})$ is *suitable for* $\mathcal{F}_{decl}$, iff there is exactly one defining expression $f(x_1, ..., x_n) = e$ in FD for each $f : \tau_1 \times ... \times \tau_n \to \tau \in \mathcal{F}_{decl}$ such that $f(x_1, ..., x_n) = e \in FD_\tau(X, \Sigma, \mathcal{F}_{decl})$. □

Apart from terms over $\Sigma \cup (\emptyset, \emptyset, \emptyset, \emptyset, \mathcal{F}_{decl})$ the so-defined expressions include basic operations as regular functions. We, however, permit basic operations over $gen(\Sigma)$-terms only. This keeps our semantics simple as we can re-use $\overset{op}{\rightsquigarrow}$ (cf. Defn. 6.2.8, page 94). Expressiveness does not suffer since variables are regular Σ-terms and, hence, values can be assigned to them using arbitrary expressions that are encapsulated in `let` constructs.

The definitions of `let` expressions and `if` expressions are straightforward. Set comprehension is covered as well. In an expression $\{e' \mid x \leftarrow e,\ e_B\}$, the term e' computes a member of the target set for a given x of the source set e if the Boolean term e_B evaluates to `True` for x. Finally, a typecast operator `cast` can be used for down-casts. This is standard in regular programming languages with subtyping (cf. casts in *JAVA*). At this point we do not further go into detail with expression syntax and well-typedness; in App. B.4, page 193, we derive well-typedness of the example in Fig. 6.1.

Before we introduce the language semantics, we want to stress that the side-effects introduced by basic operations are a source of non-determination.

Example 6.3.1 (Sources of non-determination)

Depending on the evaluation order for the parameters, the function `f`, which is defined by

$\texttt{f}(x, y) = \texttt{f}'(\texttt{create}(\texttt{c}, x), \texttt{existDObj})$
$\texttt{f}'(x, y) = y$,

returns two different values of `existDObj`. Either it evaluates to the "current" `existDObj` not containing the result of $\texttt{create}(c, x)$, or it returns $\texttt{existDObj} \cup \{\texttt{create}(c, x)\}_s$. Hence, non-determinism turns into ambiguity due to the side-effects of the basic operations.

Also, consider the following example:

$$\begin{array}{l}\texttt{f}(x, y, z) = \\ \quad \texttt{let} \quad result \quad = \quad \texttt{f}'(\texttt{create}(\texttt{c}, x), \texttt{create}(\texttt{c}, y), \texttt{create}(\texttt{c}, z)) \\ \quad \texttt{in} \quad \texttt{oid}(\texttt{snd}(result)) = \texttt{nextID}(\texttt{oid}(\texttt{fst}(result)))\end{array}$$

$$\texttt{f}'(x, y, z) = (x, z)$$

Again, $\texttt{f}(x, y, z)$ returns `True` or `False` depending on the evaluation order of the three object creation operations.

□

Table 6.6: *Expressions and function definitions over Σ, X, $\mathcal{F}_{decl}$*

Expression syntax $E_\tau(X, \Sigma, \mathcal{F}_{decl})$:

(1) **Σ-terms:**

$$\frac{t \in T_\tau(X, gen(\Sigma) \cup (\emptyset, \emptyset, \emptyset, \emptyset, \mathcal{F}_{decl}))}{t \in E_\tau(X, \Sigma, \mathcal{F}_{decl})}$$

(2) **Basic operations:**

$$\frac{op \in \mathcal{BO}p_\tau(X, gen(\Sigma))}{op \in E_\tau(X, \Sigma, \mathcal{F}_{decl})}$$

(3) **Let expressions:**

$$\frac{\begin{array}{ll} e_1 \in E_{\tau_1}(X, \Sigma, \mathcal{F}_{decl}) & \\ \quad ... & x_1 : \tau_1 \in X \\ e_n \in E_{\tau_n}(X, \Sigma, \mathcal{F}_{decl}) & \quad ... \\ e_{n+1} \in E_\tau(X, \Sigma, \mathcal{F}_{decl}) & x_n : \tau_n \in X \end{array}}{\texttt{let } x_1 = e_1 ... x_n = e_n \texttt{ in } e_{n+1} \in E_\tau(X, \Sigma, \mathcal{F}_{decl})}$$

$$\mathcal{FV}\left(\begin{array}{l}\texttt{let } x_1 = e_1 ... \\ x_n = e_n \texttt{ in } e_{n+1}\end{array}\right) = \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \backslash \{x_1\} \cup ... \cup \mathcal{FV}(e_{n+1}) \backslash \{x_1, ..., x_n\}$$

(4) **If-then-else expressions:**

$$\frac{\begin{array}{l} e_0 \in E_\tau(X, \Sigma, \mathcal{F}_{decl}) \\ e_1 \in E_\tau(X, \Sigma, \mathcal{F}_{decl}) \quad e_B \in E_{\texttt{Bool}}(X, \Sigma, \mathcal{F}_{decl}) \end{array}}{\texttt{if } e_B \texttt{ then } e_0 \texttt{ else } e_1 \ \in E_\tau(X, \Sigma, \mathcal{F}_{decl})}$$

$$\mathcal{FV}\left(\begin{array}{l}\texttt{if } e_B \texttt{ then } e_0 \\ \texttt{else } e_1\end{array}\right) = \mathcal{FV}(e_B) \cup \mathcal{FV}(e_0) \cup \mathcal{FV}(e_1)$$

(5) **Set comprehension:**

$$\frac{\begin{array}{l} e \in E_{\texttt{Set}[\tau_x]}(X, \Sigma, \mathcal{F}_{decl}) \\ x : \tau_x \in X, \\ e_B \in E_{\texttt{Bool}}(X, \Sigma, \mathcal{F}_{decl}) \\ e' \in T_{\tau'}(X, gen(\Sigma) \cup (\emptyset, \emptyset, \emptyset, \emptyset, \mathcal{F}_{decl})) \end{array}}{\{e' \mid x \leftarrow e, \ e_B\} \in E_{\texttt{Set}[\tau']}(X, \Sigma, \mathcal{F}_{decl})}$$

$$\mathcal{FV}(\{e' \mid x \leftarrow e, \ e_B\}) = (\mathcal{FV}(e_B) \cup \mathcal{FV}(e')) \backslash \{x\} \cup \mathcal{FV}(e)$$

(6) **Typecast:**

$$\frac{e \in E_{\tau'}(X, \Sigma, \mathcal{F}_{decl}) \quad \tau < \tau'}{\texttt{cast}(e, \tau) \in E_\tau(X, \Sigma, \mathcal{F}_{decl})}$$

$$\mathcal{FV}(\texttt{cast}(e, \tau)) = \mathcal{FV}(e)$$

(7) **Subtyping:**

$$\frac{e \in E_{\tau'}(X, \Sigma, \mathcal{F}_{decl}) \ \tau' < \tau}{e \in E_\tau(X, \Sigma, \mathcal{F}_{decl})}$$

Function definition syntax:

(1) **$(X, \Sigma, \mathcal{F}_{decl})$-definitions:**

$$\frac{\begin{array}{ll} x_1 : \tau_1 \in X, ..., x_n : \tau_n \in X & f : \tau_1 \times ... \times \tau_n \to \tau \in \mathcal{F}_{decl} \\ e \in E_\tau(X, \Sigma, \mathcal{F}_{decl}) & \mathcal{FV}(e) \subseteq \{x_1, ..., x_n\} \end{array}}{f(x_1, ..., x_n) \ = \ e \in FD_\tau(X, \Sigma, \mathcal{F}_{decl})}$$

Hence, different evaluation strategies may result in truly different evaluation results. We choose *strict* evaluation. As opposed to *non-strict* evaluation, this strategy evaluates *all* sub-terms first and then uses their results to evaluate the overall term. Strict evaluation applied to $\mathtt{f}(t, t')$ would, thus, return $\mathtt{existDObj} \cup \{\mathtt{create}(c, t)\}_s$. In contrast, non-strict evaluation would return $\mathtt{existDObj}$. This shows that non-strict evaluation might "swallow" operations that have an effect. We argue that this is counter-intuitive. Also, strict evaluation is in line with some relevant literature dealing with reasoning about functional programs with effects [Tal98].

The second example shows that simply evaluating all parameters first in insufficient as well. Although no side-effect can go lost, different evaluation orders result in different values of the function call. Hence, we also fix the evaluation order in the semantics.

Definition 6.3.4 (Language semantics) Given a specification $Spec := (\Sigma, Sen)$ that is a permissible extension of the basic DA such that Σ is suitable for evaluating basic operations, a type $\tau \in \mathcal{T}$, a set $\mathcal{F}_{decl}$ of function declarations suitable for Σ, a set $FD_\tau(X, \Sigma, \mathcal{F}_{decl})$ of function definitions suitable for $\mathcal{F}_{decl}$, and an expression $e \in E_\tau(X, \Sigma, \mathcal{F}_{decl})$. Then the *expression semantics for* e is defined as shown in Tab. 6.7. □

Since basic operations introduce side-effects, the language semantics carries a current state. Also, we want to facilitate object tracing. Therefore, the language semantics includes a migration sequence that contains exactly those basic state changes that have been generated in the course of the algorithm execution so far. Both, the current state and the current migration sequence can be changed by basic operations only. Respective expressions are evaluated using the $\overset{op}{\rightsquigarrow}$ operator. Expressions of other types cannot themselves cause state changes. They can be understood as means for computing the parameters for the basic operations.

$gen(\Sigma)$-terms contain no function call for any of the functions in $\mathcal{F}_{decl}$ and, hence, can be evaluated using $\overset{t}{\rightsquigarrow}$ in rule **Σ-terms 1**. In contrast, the rule **Σ-terms 2** covers terms including a function call for $f \in \mathcal{F}_{decl}$ in their top-most position. In this case, the body of f (according to FD) is evaluated. This is well-defined only because there is a function definition in FD for all functions of $\mathcal{F}_{decl}$ — FD is suitable for $\mathcal{F}_{decl}$. The rule **Σ-terms 3** covers terms that contain a function call of $\mathcal{F}_{decl}$ somewhere else than in their top-most position. We fix the evaluation order of the parameter terms in this case from left to right.

The semantics for `let` expressions and `if` expressions is straightforward. Again, we have to take care of side-effects in `let` expressions. For all i, e_i is evaluated in the result state that arises from evaluating e_{i-1}.

The typecast operator returns $\perp$ if the derived value v is not in the type domain $\tau^{\mathcal{A}}$. We do not support explicit exception handling in order to keep our semantics simple.

Set comprehension is syntactic sugar; the semantics completely relies on appropriate `let` and `if` expressions. Hence, we need not consider it explicitly when proving properties of the functional language later on. The set comprehension semantics first evaluates e. This expression represents the set that is to be traversed. Notice that we evaluate e only once and store the result in y. That way, possible side-effects occur at most once. If e yields the empty set, the whole set comprehension expression yields $\{\}$. Otherwise, a representative of y is chosen ($\mathtt{rep}(y)$). If the Boolean expression e_B evaluates to

Table 6.7: *Expression semantics for $E(X, \Sigma, \mathcal{F}_{decl})$ and $FD \subseteq FD(X, \Sigma, \mathcal{F}_{decl})$*

(1) **Σ-terms 1:**

$$\frac{t \in T_\tau(X, gen(\Sigma))\ (\mathcal{A}, \eta, t) \overset{t}{\rightsquigarrow} v}{(\mathcal{A}, \Delta, \eta, t) \overset{mig}{\rightsquigarrow} (\mathcal{A}, \Delta, v)}$$

Σ-terms 2:

$$\frac{\begin{array}{l} f(t_1, ..., t_n) \in T_\tau(X, gen(\Sigma) \cup (\emptyset, \emptyset, \emptyset, \emptyset, \mathcal{F}_{decl})) \\ f_{\overline{\tau_i} \to \tau} \in \mathcal{F}_{decl},\ f(x_1, ..., x_n) = e \in FD \\ \forall i \in \{1, ..., n\}\ \bullet\ (\mathcal{A}_i, \Delta_i, \eta, t_i) \overset{mig}{\rightsquigarrow} (\mathcal{A}_{i+1}, \Delta_{i+1}, v_i) \\ (\mathcal{A}_{n+1}, \Delta_{n+1}, \eta[x_1 \mapsto v_1]...[x_n \mapsto v_n], e) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v) \end{array}}{(\mathcal{A}_1, \Delta_1, \eta, f(t_1, ..., t_n)) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v)}$$

Σ-terms 3:

$$\frac{\begin{array}{l} f(t_1, ..., t_n) \in T_\tau(X, gen(\Sigma) \cup (\emptyset, \emptyset, \emptyset, \emptyset, \mathcal{F}_{decl}))\backslash T_\tau(X, gen(\Sigma)) \\ f_{\overline{\tau_i} \to \tau} \in \mathcal{F}_{gen(\Sigma)} \\ \forall i \in \{1, ..., n\}\ \bullet\ (\mathcal{A}_i, \Delta_i, \eta, t_i) \overset{mig}{\rightsquigarrow} (\mathcal{A}_{i+1}, \Delta_{i+1}, v_i),\ f^{\mathcal{A}^\perp_{n+1}}_{\overline{\tau_i} \to \tau}(v_1, ..., v_n) = v \end{array}}{(\mathcal{A}_1, \Delta_1, \eta, f(t_1, ..., t_n)) \overset{mig}{\rightsquigarrow} (\mathcal{A}_{n+1}, \Delta_{n+1}, v)}$$

(2) **Basic operations:**

$$\frac{op \in \mathcal{BO}p_\tau(X, gen(\Sigma))\ (\mathcal{A}, \Delta, \eta, op) \overset{op}{\rightsquigarrow} (\mathcal{A}', \Delta', v)}{(\mathcal{A}, \Delta, \eta, op) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v)}$$

(3) **Let expressions:**

$$\frac{\begin{array}{l} \texttt{let}\ x_1 = e_1 ... x_n = e_n\ \texttt{in}\ e_{n+1} \in E_\tau(X, \Sigma, \mathcal{F}_{decl}),\ (\mathcal{A}, \Delta, \eta, e_1) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2, \Delta_2, v_1) \\ \forall i \in \{2, ..., n+1\}\ \bullet\ (\mathcal{A}_2, \Delta_i, \eta[x_1 \mapsto v_1]...[x_{i-1} \mapsto v_{i-1}], e_i) \overset{mig}{\rightsquigarrow} (\mathcal{A}_{i+1}, \Delta_{i+1}, v_i) \end{array}}{(\mathcal{A}, \Delta, \eta, \texttt{let}\ x_1 = e_1 ... x_n = e_n\ \texttt{in}\ e_{n+1}) \overset{mig}{\rightsquigarrow} (\mathcal{A}_{n+2}, \Delta_{n+2}, v_{n+1})}$$

(4) **If-then-else expressions:**

$$\frac{\begin{array}{l} \texttt{if}\ e_B\ \texttt{then}\ e_0\ \texttt{else}\ e_1 \in E_\tau(X, \Sigma, \mathcal{F}_{decl}) \\ (\mathcal{A}, \Delta, \eta, e_B) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2, \Delta_2, \texttt{True}^{\mathcal{A}}),\ (\mathcal{A}_2, \Delta_2, \eta, e_0) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v) \end{array}}{(\mathcal{A}, \Delta, \eta, \texttt{if}\ e_B\ \texttt{then}\ e_0\ \texttt{else}\ e_1) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v)}$$

$$\frac{\begin{array}{l} \texttt{if}\ e_B\ \texttt{then}\ e_0\ \texttt{else}\ e_1 \in E_\tau(X, \Sigma, \mathcal{F}_{decl}) \\ (\mathcal{A}, \Delta, \eta, e_B) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', \texttt{False}^{\mathcal{A}}),\ (\mathcal{A}_2, \Delta_2, \eta, e_1) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v) \end{array}}{(\mathcal{A}, \Delta, \eta, \texttt{if}\ e_B\ \texttt{then}\ e_0\ \texttt{else}\ e_1) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v)}$$

$$\frac{\texttt{if}\ e_B\ \texttt{then}\ e_0\ \texttt{else}\ e_1 \in E_\tau(X, \Sigma, \mathcal{F}_{decl})\ (\mathcal{A}, \Delta, \eta, e_B) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', \perp)}{(\mathcal{A}, \Delta, \eta, \texttt{if}\ e_B\ \texttt{then}\ e_0\ \texttt{else}\ e_1) \overset{mig}{\rightsquigarrow} (\mathcal{A}, \Delta, \perp)}$$

(5) **Set comprehension:**

$$\frac{\begin{array}{l} \{e' \mid x \leftarrow e,\ e_B\} \in E_\tau(X, \Sigma, \mathcal{F}_{decl}), y \notin \mathcal{FV}(e) \cup \mathcal{FV}(e') \cup \mathcal{FV}(e_B) \\ (\mathcal{A}, \Delta, \eta, \texttt{let}\ y = e \\ \quad \texttt{in if}\ \ y = \{\}\ \texttt{then}\ \{\} \\ \quad\quad \texttt{else let}\ x = \texttt{rep}(y)\ \texttt{in if}\ e_B\ \texttt{then}\ \{e'\}_s \cup \{e' \mid x \leftarrow y\backslash\{x\}_s,\ e_B\} \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{else}\ \{e' \mid x \leftarrow y\backslash\{x\}_s,\ e_B\}\ ) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v') \end{array}}{(\mathcal{A}, \Delta, \eta, \{e' \mid x \leftarrow e,\ e_B\}) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v')}$$

(6) **Typecast:**

$$\frac{\begin{array}{l} \texttt{cast}(e, \tau) \in E_\tau(X, \Sigma, \mathcal{F}_{decl}) \\ (\mathcal{A}, \Delta, \eta, e) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v) \\ v \in \tau^{\mathcal{A}'} \end{array}}{(\mathcal{A}, \Delta, \eta, \texttt{cast}(e, \tau)) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v)} \qquad \frac{\begin{array}{l} \texttt{cast}(e, \tau) \in E_\tau(X, \Sigma, \mathcal{F}_{decl}) \\ (\mathcal{A}, \Delta, \eta, e) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v) \\ v \notin \tau^{\mathcal{A}'} \end{array}}{(\mathcal{A}, \Delta, \eta, \texttt{cast}(e, \tau)) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', \perp)}$$

`True` for this representative, the set comprehension expression is called recursively for the remaining set. The result of this recursive call together with $\{e'\}_s$ then form the overall result. If, however, e_B evaluates to `False` for the representative, the result of the recursive call is returned. Notice that using a fresh variable y is mandatory as, otherwise, `let` $y = e$ might unintentionally bind a variable y that occurs free in e', e. An example is shown in App. B.4, page 195.

The next theorem states that the language semantics is well-defined and, hence, indeed derives a unique value for each expression.

Theorem 6.3.1 (Language semantics is well-defined) The language semantics is well-defined, i.e., given an expression $e \in E_\tau(X, \Sigma, \mathcal{F}_{decl})$, an algebra $\mathcal{A}$, a variable assignment η, a migration sequence Δ over Σ, and a set $FD \subseteq FD(X, \Sigma, \mathcal{F}_{decl})$ of suitable function definitions for $\mathcal{F}_{decl}$, there are exactly one algebra $\mathcal{A}'$, one value $v \in \tau^{\mathcal{A}'} \cup \{\bot\}$, and a Δ' such that $(\mathcal{A}, \Delta, \eta, e)) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v)$. □

The proof can be found in App. C.4, page 222. It goes by straightforward induction on the structure of e and relies on well-definedness of $\overset{t}{\rightsquigarrow}$ and $\overset{op}{\rightsquigarrow}$.

Now, we are ready to define migration algorithms.

Definition 6.3.5 (Migration algorithms) Given a permissible extension $Spec(\Sigma, \mathcal{S}en)$ of the basic DA, such that Σ is suitable for evaluating basic operations, a set X of variables suitable for Σ, and a set of function declarations $\mathcal{F}_{decl}$ suitable for Σ. Then a *migration algorithm* $Alg(X, \Sigma, \mathcal{F}_{decl}) := (FD, f_{main})$ *over* X, Σ *and* $\mathcal{F}_{decl}$ consists of a set $FD \subseteq FD(X, \Sigma, \mathcal{F}_{decl})$ of function definitions suitable for $\mathcal{F}_{decl}$ and a main function $f_{main} \in \mathcal{F}_{decl}$. The set of all migration algorithms over X, Σ, and $\mathcal{F}_{decl}$ is denoted by $\mathcal{A}lg(X, \Sigma, \mathcal{F}_{decl})$. □

Using an explicit function f_{main} we can collect all necessary functions as a unit (algorithm). Application of an algorithm to parameters then yields the application of the main function to these parameters. A sample algorithm is provided in App. B.4, page 195; there we also derive well-formedness and parts of the semantics of the algorithm.

6.4 Summary

Since formal state changes as introduced in Sect. 3.3.2 lack practical usability we have introduced concrete implementations based of Abstract State Machines. The resulting *operations*

- `create`$(\overline{t_i})$ (object creation),
- `delete`(t) (object deletion), and
- `transform`$(t \mapsto (f, \overline{t_i}))$ (object transformation).

can be used as regular functions. The semantics translates function application to corresponding formal state changes.

These three operations have been integrated into a functional language that particularly supports set comprehension. Syntax and semantics of this programming language have been defined formally. As basic state change operations are included, the semantics

of function application yields three outputs: a resulting system state, a concrete value, and a migration sequence containing the state change history induced by the underlying function call.

We now have an expressive language available for programming migration algorithms. The semantics has been proved to be sound; generated migration sequences indeed correspond to those state changes that have been made when executing a function call. Also, the generated state change history allows for a straightforward tracing of object histories; formal preservation constraints can directly be checked if transformations are implemented using this programming language.

Chapter 7

Incorporating Graph-Based Queries

Preserving link structures can be important; the meaning of a website can — to a large extent — be determined by its browsing structure. Although automated *and* "trustworthy" preservation of link consistency is easy to postulate, it is hard to carry out, in particular, if "trustworthy" means "provably working correct". In general, references must conform to standardized languages like XPath or XQuery ([Wor03, Wor07b]); in our example, we require well-formed URIs w.r.t. [Int98]. Also, queries can appear as part of a document's content. Query semantics is determined w.r.t. graph structures, which have to be extracted from ADT specifications in our approach. As applications are manifold in our domain and graph-based queries are hard to express using FOPL formulas, we introduce a general method to semantically evaluating *and* constructing graph queries in our setting. The road map is as follows:

- In Sect. 7.1 we show how we integrate syntactic well-formedness and graph query semantics into our approach using the running example.
- Context-free grammars describe query syntax in our system. There, selected non-terminals are used to connect syntax and semantics. For this purpose, we introduce two operations called *separation* and *reduction* in Sect. 7.2.
- Query *semantics* is dealt with in Sect. 7.3. We introduce a specification scheme for graph-based query structures. There, digital objects are vertices and yield query semantics. Edges are labeled appropriately for being connected to corresponding non-terminals of the underlying context-free grammar.
- In Sect. 7.4 we combine syntax and semantics. So-called *dominated product automata* are introduced. They are constructed fully automatically and allow for evaluating and constructing graph queries that are both syntactically well-formed and semantically correct. Our method scales to a broad class of graph queries and comes with an acceptable runtime cost; we claim that this is valuable for our application domain.
- We close in Sect. 7.5 with a summary.

Our approach is adapted to standard techniques used in other related contexts like model checking [MOSS99], or XML-based query evaluation [Nev02, Chi00]. Therefore,

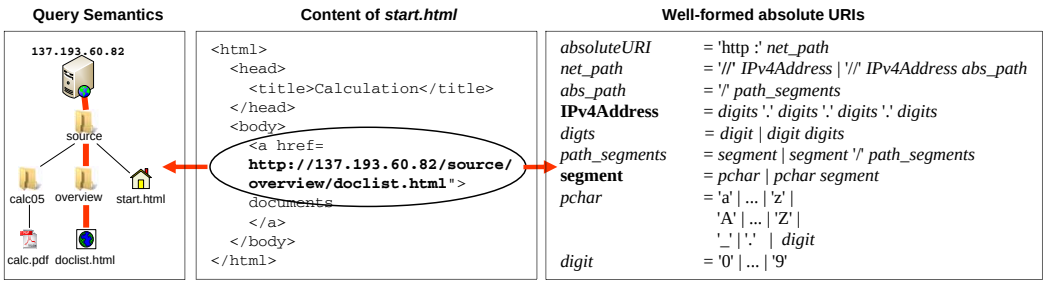


Figure 7.1: Example link and subset of absolute URIs as per *RFC 2396*

we will omit details at some points. As this chapter will be important for our case study, we mostly exemplify the formal definitions using our running example; we do not shift examples to the appendix as has been done in previous chapters.

7.1 Informal Overview

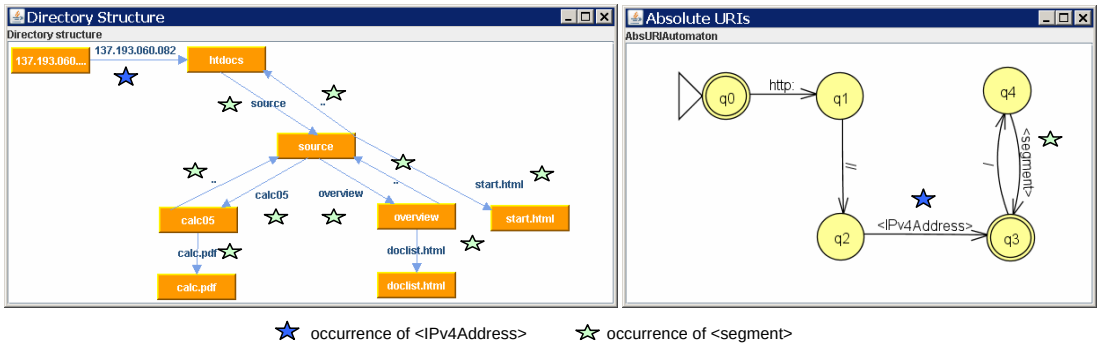
In Fig. 2.2 on page 17 we have depicted the example website "Calculation" and a permissible transformation result. Since the transformation result must conform to the *BWeb* format, the directory structure changes.

In order to integrate automated link evaluation, we have to answer the following questions: (1) How do links occur in our model? (2) How are they constrained syntactically? (3) How are they evaluated semantically? Fig. 7.1 visualizes the answers to these three questions using an example link.

In the middle part, we have listed the content of "start.html". It *contains* an absolute link to "doclist.html". In the left-hand part we have highlighted the link semantics. The link follows a *path* through the example directory structure. Both, the content of "start.html" and the directory structure that is queried by this link, are part of our website model. They are hosted by our DA as digital objects.

In the right-hand part of Fig. 7.1, we list a reduced version of the URI-reference *grammar* ([Int98]). It describes *syntactic* well-formedness of absolute URIs and, hence, carries no semantics in itself. For brevity, absolute URLs support the HTTP protocol and IPv4-addresses only.

Generally speaking, we want to evaluate graph queries that occur as part of the content of digital objects. This is challenging as our query language (FOPL) does not include graph queries directly. Therefore, we aim at combining these two worlds — DAs with formally specified data types and functionality on the one hand, and syntactic well-formedness specified by context-free grammars on the other hand. With the sample link in "start.html" we can observe some properties. First, / is purely syntactic material for separating the IP address `137.193.060.082` and the path segments `source` and `doclist.html`. Moreover, the IP address and the path segments are already part of our model for servers and directories, respectively. The IP address of a `Server` object can be obtained by the `addr`-attribute. Also, directory and document names are stored by the `name` attribute. This indicates that the non-terminals *IPv4Address* and *segment* carry *semantics* related to our model. Therefore, we have highlighted them bold in Fig. 7.1.


Directory Structure
Directory structure
137.193.060....
137.193.060.082
htdocs
source
..
source
..
calc05
overview
..
start.html
calc05
overview
start.html
calc.pdf
doclist.html
calc.pdf
doclist.html
Absolute URIs
AbsURIAutomaton
q0
http:
q1
q4

=
/
<IPv4Address>
q2
q3
occurrence of <IPv4Address>
occurrence of

Figure 7.2: Example automaton for absolute URIs and a directory structure

The key idea of our approach is to separate concerns. First, we specify *query structures* (similar to the one in the left-hand part of Fig. 7.1) using an appropriate graph specification scheme, which is translated to FOPL formulas. There, digital objects are vertices and constitute *query semantics*. Edges labels carry words that can be assigned to non-terminals of the underlying grammar.

Second, we translate well-formedness rules to appropriate Finite State Automata (FSA, [HMU06]). This translation is standard. Using automata we, however, do not need to incorporate parsers explicitly. In this context parse trees are unnecessary anyway.

Third, we combine both. There, the "semantic" non-terminal symbols are connecting points. The resulting automaton basically evaluates both structures in parallel. The benefits are twofold: (1) We have a procedure for *checking well-formedness* of graph queries and *generating well-formed* graph queries. (2) We have a mechanism for *evaluating the semantics* of graph queries and for *constructing queries that have a desired semantics*. There, semantics is determined by components that are hosted by a DA. Here, this comprises directories and files. An example is shown in Fig. 7.2.

The left-hand part directly corresponds to the query structure of Fig. 7.1 (left-hand side). Vertices are labeled with the server IP and directory/file name, respectively, of those entities they represent. Edge labels carry IP addresses and file or directory names. In particular, the server with IP address 137.193.060.082 has been used for the running example. It is connected to its source directory "htdocs" (standard source directory for Apache Web Servers) via an edge labeled with its IP address. Navigation in the directory structure is possible from directories to directories (bidirectional) and from directories to files. The special string ".." is used for backwards navigation between directories (cf. [Int98]).

In the right-hand part we show an FSA that recognizes absolute URIs. The starting state is marked by a triangle. Final states are marked by double circles. We use two different types of transitions. Those labeled `http` :, //, and / fire on recognition of the respective symbol. The other transitions ($\langle IPv4Address \rangle$, $\langle segment \rangle$) represent the corresponding non-terminals of the URI reference grammar (marked bold in the right-hand part of Fig. 7.1). In order to demonstrate the correlation between these *hierarchical transitions* and the edges of the query graph, we have marked the corresponding edges by differently colored stars. The edge label "`137.193.060.082`" is a well-formed IP address

that can be produced by the non-terminal $IPv4Address$ of the URI reference grammar. Also, the other edge labels of the query graph are well-formed path segments, which are produced by the non-terminal *segment*.

When evaluating links, we run through both structures in parallel; underlying techniques are related to product automata. The link

`http://137.193.060.082/source/overview/doclist.html`,

e.g., starts at state q_0 of the URI automaton. The initial semantics (also *semantic object*) is set to the server object in the query structure (vertex labeled with "137.193.060.082"). Next, the URI automaton recognizes `http:` and proceeds from state q_0 to state q_1. Query semantics remains unchanged; there is no corresponding edge in the respective graph structure. Similarly, the URI automaton switches to q_2 in the next step as it recognizes //. The query semantics still yields the server object. Next, `137.193.060.082` is recognized. Since this word can be produced by $IPv4Address$, the URI automaton switches to state q_3. Also, the query semantics changes to "htdocs"; there is an edge labeled "137.193.060.082" to the vertex representing this directory. So far we have recognized the string `http://137.193.060.082`. This is already a valid URI (q_3 is a final state in the URI automaton). The semantics of this URI is the directory "htdocs". Evaluation proceeds accordingly until the whole link has been processed. The final semantic object yields "doclist.html", which is indeed the intended link target.

The overall procedure is general enough for being applied to other query languages like selectors in Cascading Style Sheets (CSS, [Wor07a]) or XPath ([Wor03]) as well. The underlying theories are well-studied and particularly allow for proofs w.r.t. syntactic and semantic link correctness. This meets our claim for high trustworthiness and entails acceptable runtime costs (see case study in Chap. 8). We argue that this is beneficial and justifies the effort. Particularly, when processing large object collections checking this kind of property cannot be done by hand.

In the following sections we provide details. We start with query syntax, proceed with query semantics, and finally integrate both.

7.2 Query Syntax — Integrating Regular Languages

In official specifications or standards, syntactic well-formedness rules are often given by *grammars* (as indicated by the URI-reference grammar). Compared to other description methods (like Finite State Automata) grammars tend to be more human-readable. The names of non-terminals like $IPv4Address$ indicate what is "meant"; we take grammars as starting point and show how to apply them in our context.

We denote context-free grammars by $G := (N, \Omega, P, S)$, where, N denotes non-terminals, Ω denotes terminals, and P and S denote the set of productions and the start symbol, respectively. In order to avoid confusions, we use Ω instead of Σ or T (which are most widely used in the literature); the latter denote signatures and terms already. Since alphabets of FSA will also be denoted by Ω later on, this even emphasizes the connections.

We denote productions by $A \to \mathtt{a}B\mathtt{c}$, where we exclude ϵ productions (ϵ denotes empty symbol sequences). By convention, italics like A, B denote non-terminals. Termi-

nals are denoted like $\mathtt{a}, \mathtt{c}$. Greek letters α, β, γ denote elements of $(\Omega \cup N)^*$. Derivation in G is denoted by $\overset{G}{\Rightarrow}$ (one-step) and $\overset{G}{\Rightarrow}^*$ (iterated), respectively, and $L(G)$ denotes the language generated by G.

Since we aim at translating grammars into corresponding FSA, we restrict ourselves to those grammars that can equivalently be represented by a right-linear (equivalently: left-linear) grammar; they produce exactly those languages that can be recognized by FSA [HMU06]. Right-linear grammars may contain productions of the form $A \to \mathtt{a}B$ or $A \to \mathtt{a}$ only. Notice that the core of today's standard query languages like XPath, XQuery ([Wor07b]), or CSS is regular. This is already indicated by the fact that they query *graph paths*. Paths are defined sequentially. This can be reflected by productions that work sequentially (from left to right or from right to left).

Now, we switch to those non-terminals that we consider to "carry semantics". The basic idea is to separate them from the original grammar. After that we construct a *hierarchical* FSA that exactly recognizes the language that is produced by the original grammar. It has two different types of transitions as shown in the right-hand part of Fig. 7.2. The top-level automaton carries the state transitions related to the purely syntactic material of the URI references (like $\mathtt{http}:, /$). The *hierarchical* transitions carry those automata that are generated for the "semantic" non-terminals. Details follow in Sect. 7.4.

Definition 7.2.1 (Non-terminal reachability) Given a grammar $G := (N, \Omega, P, S)$. Then *non-terminal reachability* $\mathit{reach}_G \subseteq N \times N$ *in* G is defined by

$$\mathit{reach}_G := \{(A, B) \mid \exists \alpha, \beta \bullet A \to \alpha B \beta \in P\}^*$$

Moreover, $\mathit{reach}_G(A) := \{B \mid (A, B) \in \mathit{reach}_G\}$. □

Given a non-terminal A, reach_G assigns all non-terminals to A that can be reached from A by a sequence of productions in P; * denotes the reflexive-transitive closure.

Using *reach*, decomposition of grammars is defined as follows.

Definition 7.2.2 (Reduction and separation) Given a grammar $G := (N, \Omega, P, S)$ and a non-terminal $A \in N$. Then *the reduction* $\mathit{red}(A, G)$ *of* G *to* A is defined by

$$\begin{aligned}\mathit{red}(A, G) := (\ & \mathit{reach}_G(A),\\ & \Omega,\\ & \{X \to \alpha \mid X \to \alpha \in P, X \in \mathit{reach}_G(A)\},\\ & A).\end{aligned}$$

The *separation* $\mathit{sep}(A, G)$ *of* A *from* G is defined if $A \neq S$. In this case, it yields

$$\begin{aligned}\mathit{sep}(A, G) := (\ & N \backslash \{A\},\\ & \Omega \cup \{A\},\\ & P \backslash \{A \to \alpha \mid A \to \alpha \in P\},\\ & S)\end{aligned}$$

□

Reduction and separation decompose a grammar. The reduction of G to A defines the "sub-grammar" of G with start symbol A. In particular, the language of $\mathit{red}(A, G)$ contains all words that can be produced from A in G. When separating G and A, we shift A to the terminals and remove all productions for A. It is easy to see that both constructions yield well-defined grammars. In particular, $S \in N$ and $N \cap \Omega = \emptyset$ hold.

Example 7.2.1 (Reduction and separation)

Denote the grammar in Fig. 7.1 by G_{aURI}, $red(IPv4Address, G_{aURI})$ consists of:

$N := \{\ IPv4Address, digits, digit\}$
$\Omega := \{\ \_, ., \mathtt{a}, ..., \mathtt{z}, \mathtt{A}, ..., \mathtt{Z}, \mathtt{0}, ..., \mathtt{9}, \mathtt{http}\ :, //, /\}$
$P := \{\ IPv4Address \rightarrow digits\ .\ digits\ .\ digits\ .\ digits$
$\quad digits \rightarrow digit \mid digit\ digits$
$\quad digit \rightarrow \mathtt{0} \mid ... \mid \mathtt{9}\ \}$
$S := IPv4Address$

The new grammar accepts IP addresses.
$sep(IPv4Address, G_{aURI})$ consists of:

$N := \{\ absoluteURI, net\_path, abs\_path,$
$\quad digits, path\_segments, segment, pchar, digit\}$
$\Omega := \{\ \mathtt{IPv4Address}, \_, ., \mathtt{a}, ..., \mathtt{z}, \mathtt{A}, ..., \mathtt{Z}, \mathtt{0}, ..., \mathtt{9}, \mathtt{http}\ :, //, /\}$
$P := \{\ absoluteURI \rightarrow \mathtt{http}\ :\ net\_path$
$\quad net\_path \rightarrow //\ \mathtt{IPv4Address} \mid //\ \mathtt{IPv4Address}\ abs\_path$
$\quad abs\_path \rightarrow /\ path\_segments$
$\quad digits \rightarrow digit \mid digit\ digits$
$\quad path\_segments \rightarrow segment \mid segment\ /\ path\_segments$
$\quad segment \rightarrow pchar \mid pchar\ segment$
$\quad pchar \rightarrow \_ \mid . \mid \mathtt{a} \mid ... \mid \mathtt{z} \mid \mathtt{A} \mid ... \mid \mathtt{Z} \mid digit$
$\quad digit \rightarrow \mathtt{0} \mid ... \mid \mathtt{9}\ \}$
$S := absoluteURI$

The former non-terminal $IPv4Address$ has been shifted to Ω. Also, the production $IPv4Address \rightarrow digits...$ has been removed according to Defn. 7.2.2.

□

Notice that reduction preserves right-linearity. Separation does not preserve right-linearity directly. However, the result can easily be transformed to an equivalent right-linear grammar. The non-terminal A in question always occurs at the right-most position in a production $B \rightarrow \alpha A$, where $\alpha \in \Omega$. As A has been shifted to the non-terminals, the new production can be turned into a right-linear one by introducing a new non-terminal X, replacing $B \rightarrow \alpha A$ by $B \rightarrow \alpha X$, and adding $X \rightarrow A$.

The following composition operation is inverse to *sep* and *red*.

Definition 7.2.3 (Grammar composition) Given two grammars $G := (N, \Omega, P, S)$ and $G' := (N', \Omega', P', S')$. Then G and G' are *composable*, iff $\Omega \cap N' = \{S'\}$, $\Omega' \cap N = \emptyset$, and for all $A \in N \cap N'$ it is true that

$$\{X \rightarrow \alpha Y \beta \mid X \rightarrow \alpha Y \beta \in P', \alpha, \beta \in (N' \cup \Omega')^*, X = A \vee Y = A\} =$$
$$\{X \rightarrow \alpha Y \beta \mid X \rightarrow \alpha Y \beta \in P, \alpha, \beta \in (N \cup \Omega)^*, X = A \vee Y = A\}.$$

The *composition* $G; G'$ *of* G *and* G' is defined as follows:

$G; G' := (\ N \cup N',$
$\quad (\Omega \cup \Omega') \backslash \{S'\},$
$\quad P \cup P',$
$\quad S).$

□

The productions of G and G' must be equal for shared non-terminals. In our example both, $red(IPv4Address, G_{aURI})$ and $sep(IPv4Address, G_{aURI})$, contain *digit* and

digits. Since they equally contain all related productions as well, both grammars are composable.

Since the start symbol S' of G' occurs in Ω, the production sets P and P' can simply be joined. Both production sets are valid productions for $G; G'$. According to the next lemma, separating G and A and composing them again yields a grammar that generates exactly the same language as G.

Lemma 7.2.1 (Composition inverse to decomposition) Given a grammar $G := (N, \Omega, P, S)$ and a non-terminal $A \in N$ such that separation of A and G is defined (recall: $A \neq S$). Then $L(sep(A, G); red(A, G)) = L(G)$. □

The proof can be found in App. C.5, page 223. Separating non-terminals from the original grammar will be particularly important when combining syntax and semantics.

Notice that Lemma 7.2.1 holds for repeated decomposition as well. As an example,

$$L(G) = L(sep(A, sep(B, G)); red(B, G); red(B, G)).$$

This situation frequently occurs. In our example, $IPv4Address$ and *segment* are affected. We conclude this section by listing the productions of

$$sep(segment, sep(IPv4Address, G_{aURI}))$$

in order to demonstrate the achievement:

$$\begin{array}{ll} absoluteURI & \rightarrow \texttt{http} : net\_path \\ net\_path & \rightarrow \texttt{//}\ \texttt{IPv4Address} \mid \texttt{//}\ \texttt{IPv4Address}\ abs\_path \\ abs\_path & \rightarrow \texttt{/}\ path\_segments \\ path\_segments & \rightarrow \texttt{segment} \mid \texttt{segment}\ \texttt{/}\ path\_segments \end{array}$$

First, it is easy to see that this grammar has an equivalent right-linear grammar. Hence, it can be translated into a corresponding FSA. This automaton is equivalent to the one shown in the right-hand part of Fig. 7.2. Moreover, we have reduced G_{aURI} such that it merely produces syntactic material like the separators // and /. Those non-terminals that carry semantics and are connected to the graph structure on the left-hand side of Fig. 7.2 ($IPv4address$ and *segment*) are handled as regular terminal symbols. They serve as connecting points to the respective grammars that have been "sourced out". This will be important when constructing dominated product automata in Sect. 7.4.

7.3 Query Semantics — Specifying Graph Structures

The running example shows that query semantics is closely related to the content of DAs in our setting. Links, e.g., point to digital objects that are organized into hierarchical directory structures. Hence, we specify how the related graph structures can be *extracted* from the formal datatypes and functionalities that are part of a DA.

The query graph in the left-hand part of Fig. 7.2 indicates that we use labeled, directed graphs. Yet, we explicitly store permissible starting points for queries and permissible query results; html links, e.g., may lead to directories and files only.

Definition of vertex set V w.r.t. variables x_i				
No.	**Variable**	**Condition for** $x_i \in V$	**Condition for** $x_i \in V^I$	**Condition for** $x_i \in V^F$
1	$x_1 : \tau_1$	$\phi_V^1(x_1)$	$\phi_{V^I}^1(x_1)$	$\phi_{V^F}^1(x_1)$
...	...	...	...	...
n	$x_n : \tau_n$	$\phi_V^n(x_n)$	$\phi_{V^I}^n(x_n)$	$\phi_{V^F}^n(x_n)$
Definition of edge set E w.r.t. variables x_i^s, l_i, x_i^t				
No.	**Variables**	**Label term over** (x_i^s, x_i^t)	**Condition for** $(x_i^s, l_i, x_i^t) \in E$	—
1	$x_1^s : \tau_1^s,\ l_1 : \tau_1^l,\ x_1^t : \tau_1^t$	$t_L^1(x_1^s, x_1^t)$	$\phi_E^1(x_1^s, l_1, x_1^t)$	—
...	...	...	...	—
m	$x_m^s : \tau_m^s,\ l_m : \tau_m^l,\ x_m^t : \tau_m^t$	$t_L^m(x_m^s, x_m^t)$	$\phi_E^m(x_m^s, l_m, x_m^t)$	—

Figure 7.3: Specification scheme for graph-based query structures

In the following we will *describe* query structures formally by means of the *specification scheme* depicted in Fig. 7.3. Vertices in the resulting structures represent query semantics. First, we specify those objects that belong to the vertex set. These objects have to satisfy *one* of the imposed semantic conditions $\phi_V^1, ..., \phi_V^n$. There, we use variables $x_i : \tau_i$ as place holders; they are universally-quantified implicitly.

Similarly, $t_L^1(x_1^s, x_1^t), ..., t_L^m(x_m^s, x_m^t)$ and $\phi_E^1(x_1^s, l_1, x_1^t), ..., \phi_E^m(x_m^s, l_m, x_m^t)$ specify different variants of edges that can occur in the query graph. We use *label terms* $t_L^i(x_i^s, x_i^t)$ that return a set of permissible labels. That way graph structures can be generated efficiently; only labels occurring in these sets are permitted. The formal semantics of the above scheme is introduced in the following definition.

Definition 7.3.1 (Graph specification scheme, resulting graph) Given a specification $Spec := (\Sigma, Sen)$, $\Sigma := (\mathcal{T}, <, \mathcal{P}, \mathcal{C}, \mathcal{F})$, that is a permissible extension of the basic DA, a label type $\texttt{String} \in \mathcal{T}$, a set X of variables suitable for Σ, and a Σ-algebra $\mathcal{A}$. A *graph specification* $S_\mathcal{G} := (S_\mathcal{G}^V, S_\mathcal{G}^E)$ *over* Σ consists of

- a set of *vertex specifications*

$$S_\mathcal{G}^V := \{(x_1 : \tau_1, \phi_V^1, \phi_{V^I}^1, \phi_{V^F}^1), ..., (x_n : \tau_n, \phi_V^n, \phi_{V^I}^n, \phi_{V^F}^n)\},$$

  where for all $1 \leq i \leq n$ it is true that

  (1) $x_i : \tau_i \in X$,

  (2) $\tau_i < \texttt{DObj}$, and

  (3) $\mathcal{FV}(\phi_V^i) \cup \mathcal{FV}(\phi_{V^I}^i) \cup \mathcal{FV}(\phi_{V^F}^i) \subseteq \{x_i : \tau_i\}$,

- a set of *edge specifications*

$$S_\mathcal{G}^V := \{(x_1^s : \tau_1^s, l_1 : \tau_1^l, x_1^t : \tau_1^t, t_L^1, \phi_E^1), ..., (x_m^s : \tau_m^s, l_m : \tau_m^l, x_m^t : \tau_m^t, t_L^m, \phi_E^m)\},$$

  where for all $1 \leq i \leq m$ it is true that

  (1) $x_i^s : \tau_i^s, x_i^t : \tau_i^t, l_i : \tau_i^l \in X$,

  (2) $\tau_i^s, \tau_i^t < \texttt{DObj}$ and $\tau_i^l < \texttt{String}$,

  (3) $t_L^i : \texttt{Set}[\tau_i^l]$,

(4) $\mathcal{FV}(t_L^i) \subseteq \{x_i^s : \tau_i^s, x_i^t : \tau_i^t\}$, and
(5) $\mathcal{FV}(\phi_E^i) \subseteq \{x_i^s : \tau_i^s, l_i : \tau_i^l, x_i^t : \tau_i^t\}$.

The *graph-based query structure* $\mathcal{G}(S_\mathcal{G}, \mathcal{A}) := (V, E, V^I, V^F)$ *resulting from* $S_\mathcal{G}$ *in* $\mathcal{A}$ is determined by the following components:

$$
\begin{array}{lll}
V := & \{v & \mid (x : \tau, \phi_V, \phi_{V^I}, \phi_{V^F}) \in S_\mathcal{G}^V, \\
 & & v \in \tau^\mathcal{A}, \mathcal{A} \models \phi_V[\{x \mapsto v\}]\} \\
E := & \{(v^s, \mathcal{V}^\mathcal{A}[\![t^l]\!], v^t) & \mid (x^s : \tau^s, l : \tau^l, x^t : \tau^t, t_L, \phi_E) \in S_\mathcal{G}^E, \\
 & & v^s \in (\tau^s)^\mathcal{A}, v^t \in (\tau^t)^\mathcal{A}, t^l \in GT_{\tau^l}(\Sigma), \\
 & & \mathcal{A} \models t^l \in t_L \wedge \phi_E[\{x^s \mapsto v^s, x^t \mapsto v^t, l \mapsto \mathcal{V}^\mathcal{A}[\![t^l]\!]\}]\} \\
V^I := & \{v & \mid (x : \tau, \phi_V, \phi_{V^I}, \phi_{V^F}) \in S_\mathcal{G}^V, \\
 & & v \in \tau^\mathcal{A}, \mathcal{A} \models \phi_V \wedge \phi_{V^I}[\{x \mapsto v\}]\} \\
V^F := & \{v & \mid (x : \tau, \phi_V, \phi_{V^I}, \phi_{V^F}) \in S_\mathcal{G}^V, \\
 & & v \in \tau^\mathcal{A}, \mathcal{A} \models \phi_V \wedge \phi_{V^F}[\{x \mapsto v\}]\} \\
L := & \texttt{String}^\mathcal{A}. &
\end{array}
$$

□

As usual, V, E, and L denote the vertex set, edge set, and labels of the resulting graph; V^I and V^F denote permissible initial and final vertices as explained before. Also, we require Σ to contain a label type `String`. In general this type can be arbitrary. Calling it `String` however, we already suggest that we use strings later on.

Formal graph specifications directly realize the scheme in Fig. 7.3. Vertex and edge specifications are provided by tuples starting with the respective variable(s). In $S_\mathcal{G}^V$, e.g., all tuples include a variable $x : \tau$ in the first position. According to the constraints imposed in the definition above, all formulas occurring in this tuple may include no other free variables than x. This is similar with edge specifications in $S_\mathcal{G}^E$. The semantics of graph specifications is given by a graph-based query structure that carries objects as states.

Graph-based query structures can easily be translated into equivalent FSA. Notice that we particularly support *multi-graphs*, i.e., such containing differently labeled edges. In the context of XML and XPath queries, e.g., direct sub-nodes of a given XML node n may be addressed by the *child*-predicate or by their node names.

Example 7.3.1 (Graph specification for the running example)

In Fig. 7.4 we show the full specification for the query graph in Fig. 7.2. The top-most line of the vertex specification $S_\mathcal{G}^V(aURI)$ introduces a variable x_1 : `Server` followed by a `True` entry. It states that all objects of type `Server` are members of the resulting vertex set V. Moreover, all servers are starting points for query evaluation (next `True` entry), but never query results (`False`-entry). Analogously, the other two lines introduce directories and documents as permissible vertices. Yet queries cannot start there. In fact, objects of type `Dir` or `Doc` yield query semantics (last `True` entry of each line). Putting all together, the components V, V^I, V^F of the resulting graph-based query structure yield

$$
\begin{array}{lll}
V & := & \texttt{Server}^\mathcal{A} \cup \texttt{Dir}^\mathcal{A} \cup \texttt{Doc}^\mathcal{A} \\
V^I & := & \texttt{Server}^\mathcal{A} \\
V^F & := & \texttt{Dir}^\mathcal{A} \cup \texttt{Doc}^\mathcal{A}
\end{array}
$$

Fig. 7.2 indicates that we have four different variants of edges, namely those between a server and a directory, between a directory and a file, and two variants between two

Definition of vertex set V w.r.t. variables x_i				
No.	**Variable**	**Condition for** $x_i \in V$	**Condition for** $x_i \in V^I$	**Condition for** $x_i \in V^F$
1	$x_1 : \texttt{Server}$	True	True	False
2	$x_2 : \texttt{Dir}$	True	False	True
3	$x_3 : \texttt{Doc}$	True	False	True

Definition of edge set E w.r.t. variables x_i^s, l_i, x_i^t			
No.	**Variables**	**Label term over** (x_i^s, x_i^t)	**Condition for** $(x_i^s, l_i, x_i^t) \in E$
1	$x_1^s : \texttt{Server},\ l_1 : \texttt{String},\ x_1^t : \texttt{Dir}$	$\{\texttt{addr}(x_1^s)\}_s$	$x_1^t = \texttt{srcDir}(x_1^s)$
2	$x_2^s : \texttt{Dir},\ l_2 : \texttt{String},\ x_2^t : \texttt{Dir}$	$\{\texttt{name}(x_2^t)\}_s$	$x_2^t \in \texttt{subDirs}(x_2^s)$
3	$x_3^s : \texttt{Dir},\ l_3 : \texttt{String},\ x_3^t : \texttt{Dir}$	$\{\text{“..“}\}_s$	$x_3^s \in \texttt{subDirs}(x_3^t)$
4	$x_4^s : \texttt{Dir},\ l_4 : \texttt{String},\ x_4^t : \texttt{Doc}$	$\{\texttt{name}(x_4^t)\}_s$	$x_4^t \in \texttt{subDocs}(x_4^s)$

Figure 7.4: Specification $S_{\mathcal{G}}(aURI)$ of the query structure for absolute URIs

directories (downwards and upwards navigation). The edge specifications in Fig. 7.4 are a one-to-one realization thereof. In the first line we introduce variables of type `Server`, `Dir` and `String`. The corresponding label term states that edges of this type may carry the `addr`-attribute of their source object only. The last condition assures that edges of this type are set only, if the edge's target is the source directory belonging to the server object that is the edge's source. In that case, the edge label is automatically set to the server's IP (no further conditions on the label are given). Analogously, lines two to four specify downwards/upwards navigation in directory structures.

The edge set E of the resulting graph-based query structure is determined by

$$
\begin{aligned}
E \;:=\; & \{(\mathcal{V}^{\mathcal{A}}[\![t^s]\!], \mathcal{V}^{\mathcal{A}}[\![t^l]\!], \mathcal{V}^{\mathcal{A}}[\![t^t]\!]) \\
& \quad \mid\; t^s \in GT_{\texttt{Server}}(\Sigma), t^l \in GT_{\texttt{String}}(\Sigma), t^t \in GT_{\texttt{Dir}}(\Sigma), \\
& \qquad \mathcal{A} \models t^l \in \{\texttt{addr}(t^s)\}_s \wedge t^t = \texttt{srcDir}(t^s)\} \cup \\
& \{(\mathcal{V}^{\mathcal{A}}[\![t^s]\!], \mathcal{V}^{\mathcal{A}}[\![t^l]\!], \mathcal{V}^{\mathcal{A}}[\![t^t]\!]) \\
& \quad \mid\; t^s \in GT_{\texttt{Dir}}(\Sigma), t^l \in GT_{\texttt{String}}(\Sigma), t^t \in GT_{\texttt{Dir}}(\Sigma), \\
& \qquad \mathcal{A} \models t^l \in \{\texttt{name}(t^t)\}_s \wedge t^t \in \texttt{subDirs}(t^s)\} \cup \\
& \{(\mathcal{V}^{\mathcal{A}}[\![t^s]\!], \mathcal{V}^{\mathcal{A}}[\![t^l]\!], \mathcal{V}^{\mathcal{A}}[\![t^t]\!]) \\
& \quad \mid\; t^s \in GT_{\texttt{Dir}}(\Sigma), t^l \in GT_{\texttt{String}}(\Sigma), t^t \in GT_{\texttt{Dir}}(\Sigma), \\
& \qquad \mathcal{A} \models t^l \in \{\text{“..“}\}_s \wedge t^s \in \texttt{subDirs}(t^t)\} \cup \\
& \{(\mathcal{V}^{\mathcal{A}}[\![t^s]\!], \mathcal{V}^{\mathcal{A}}[\![t^l]\!], \mathcal{V}^{\mathcal{A}}[\![t^t]\!]) \\
& \quad \mid\; t^s \in GT_{\texttt{Dir}}(\Sigma), t^l \in GT_{\texttt{String}}(\Sigma), t^t \in GT_{\texttt{Doc}}(\Sigma), \\
& \qquad \mathcal{A} \models t^l \in \{\texttt{name}(t^t)\}_s \wedge t^t \in \texttt{subDocs}(t^t)\}.
\end{aligned}
$$

The order of the disjunction terms directly corresponds to the order of the specification lines in Fig. 7.4.

□

Notice that these specifications have to be set up only once and scale to all system states. In the course of system evolution, the resulting graphs grow as well. The formal underpinning of these schemes allows us to prove properties about the specified graphs. In our example, e.g., all query results are either the source directory of a server object or are recursively contained in a source directory of a server object. We could also prove general properties of the result graph like uniqueness of shortest paths or loop- and circle-freeness (the latter only in the absence of edges labeled with “..“). We omit details for brevity.

7.4 Automated Query Evaluation and Construction

In the following we need generic automata. Formally, the theory of hierarchical automata [AKY99, MLS97] is suitable for our purposes. Yet the full theory is quite involved and largely unnecessary in our context. Therefore, we use simple word-accepting, non-concurrent (no parallelism) hierarchical automata, where *transitions* can carry automata.

There, we assume a global symbol set $\widehat{\Omega}$ that is closed under composition ; with unit ϵ (often referred to as the *free monoid* in the literature). In particular, $\widehat{\Omega}^* \subseteq \Omega$; we do not distinguish between symbols and words and tokenizing is part of our notion of word acceptance. This requires only little extra-effort but eases datatype definitions in practice; we can use $(\texttt{String}, ++)$ for $(\widehat{\Omega}, ;)$. As usual, we denote words by u, v, w.

In line with the relevant literature, we denote hierarchical automata (HA) by tuples $HA := (\Omega, Q, Q^I, Q^F, HA^s, \delta)$, where the components represent

- a set of *symbols* $\Omega \subseteq \widehat{\Omega}$,
- a set of *states* Q,
- a set of *initial states* $Q^I \subseteq Q$,
- a set of *final states* $Q^F \subseteq Q$,
- a set of *sub-automata* HA^s, $HA^s \cap \Omega = \emptyset$,
- a set of *transitions* $\delta \subseteq Q \times \Omega \cup HA^s \times Q$.

Without loss of generality we exclude ϵ-transitions.

Later on, we will sometimes need the *extended symbol set* $symbols^e(HA)$ *of* HA, which carries all symbols of HA and all *words* that can be composed of symbols of its sub-automata, i.e.

$$symbols^e(HA) : HA_\Omega \cup (\bigcup_{a \in HA^s} symbols^e(a)^*).$$

Transitions carrying elements of HA^s are called *hierarchical*. A hierarchical FSA is called *basic*, iff it has no hierarchical transitions and HA^s is empty. Basic HA correspond to regular FSA (cf. [HMU06]). Query graphs of the last section can be directly translated into basic HA; we do not provide a separate construction and treat query graphs as HA in the following.

Example 7.4.1 (Example HA for absolute URIs)

The automaton HA_{aURI} for absolute URIs is given by the following components (cf. Fig. 7.2):

$$\begin{array}{lcl}
\Omega_{aURI} & := & \{\text{“}\texttt{http}:\text{“}, \text{“}//\text{“}, \text{“}/\text{“}\} \\
Q_{aURI} & := & \{q_0, q_1, q_2, q_3, q_4, q_5\} \\
Q^I_{aURI} & := & \{q_0\} \\
q^F_{aURI} & := & \{q_3, q_4, q_5\} \\
HA^s_{aURI} & := & \{HA_{IPv4}, HA_{seg}\} \\
\delta_{aURI} & := & \{(q_0, \text{“}\texttt{http}:\text{“}, q_1), (q_1, \text{“}//\text{“}, q_2), (q_2, HA_{IPv4}, q_3), \\
 & & \quad (q_3, \text{“}/\text{“}, q_4), (q_4, HA_{seg}, q_5), (q_5, \text{“}/\text{“}, q_4)\}
\end{array}$$

It has two sub-automata for producing IP-addresses and path segments, respectively.

□

There are two ways transitions can fire in HA. When evaluating

$$\text{“}\texttt{http://137.193.060.082/source}\text{“}$$

in HA_{aURI}, “`http` : “ and “//“ are recognized by basic transitions like in regular FSA. “`137.193.60.82`“, however, is recognized by HA_{IPv4}, which is a sub-automaton of HA_{aURI}. The corresponding transition (q_2, HA_{IPv4}, q_3) fires because HA_{IPv4} accepts “`http` : “; the notion of word acceptance is recursive. As we need it when proving formal correctness of our constructions, we introduce it in the next definition.

Definition 7.4.1 (Runs, word acceptance) Given a hierarchical automaton $HA := (\Omega, Q, Q^I, Q^F, HA^s, \delta)$ and a word $w = w_1...w_n$. A *run* r *of* HA *over* w is a sequence $q_0 w_0 q_1 ... w_n q_{n+1}$ *of permissible steps* $q_i w_i q_{i+1}$ *in* HA $(1 \leq i \leq n)$. A step $q_i w_i q_{i+1}$ is *permissible in* HA, iff

- $q_i \in Q, q_{i+1} \in Q$, and
- there is a transition $(q_i, l, q_{i+1}) \in \delta$ that *can fire on* w_i, i.e., either
  - $w_i \in \Omega$ and $l = w_i$ or
  - $w_i \in \Omega_l^*$, $l \in HA^s$ and there is an *accepting run* of l over w_i.

A run r is *accepting*, iff additionally $q_0 \in Q^I$ and $q_{n+1} \in Q^F$. □

The language of HA is denoted by $L(HA)$. Also, we denote the corresponding acceptance relation by $accepts(HA, w)$. A hierarchical automaton HA accepts sequences of symbols that consist of symbols of HA or of words over symbols of its sub-automata. In the basic case, this directly corresponds to runs of non-hierarchical FSA.

Example 7.4.2 (Accepting runs)

HA_{aURI} accepts “`http://137.193.060.082/source`“ by

$$q_0\texttt{http}:q_1//q_2\texttt{137.193.60.82}q_3/q_4\texttt{source}q_5$$

provided that HA_{IPv4} accepts `137.193.60.82` and HA_{seg} accepts `source`. This is true in our example.

□

The notion of acceptance given in Defn. 7.4.1 is, in general, non-deterministic. First, there may be different transitions that can fire on w in a given state q. Second, hierarchical transitions may lead to sequential non-determinism. Given states $Q^I = \{q_0\}, Q^F = \{q_1, q_2\}$ and transitions (q_0, a, q_1) and $(q_1, /, q_2)$ such that a is an automaton with

$$L(a) = \{\text{“}\texttt{137.193.60.82}\text{“}, \text{“}\texttt{137.193.60.82/}\text{“}\}.$$

Then there $q_0\texttt{137.193.60.82}q_1/q_2$ and $q_0\texttt{137.193.60.82}/q_1$ are two different accepting runs for `137.193.60.82/`; these runs end in different states. Since states carry semantics later on, we will particularly be concerned with *deterministic* HA.

Definition 7.4.2 (Deterministic HA) A hierarchical automaton HA is

(1) *branching deterministic*, iff for all $q \in Q_{HA}$, words w over $symbols^e(HA)$ and segmentations $w = w_1 w_2$ there is at maximum one state $q' \in Q_{HA}$ and one transition $(q, l, q') \in \delta_{HA}$ that can fire on w_1.

(2) *sequentially-deterministic*, iff for all $q \in Q_{HA}$ and words w over $symbols^e(HA)$ there is at maximum one segmentation $w = w_1 w_2$ such that there is a state $q' \in Q_{HA}$ and a transition $(q, l, q') \in \delta_{HA}$ that can fire on w_1.

(3) *deterministic*, iff it is branching deterministic and sequentially deterministic.

□

Again, the definition reduces to the standard definition in case of basic HA. Branching determinism resolves the first source of non-determinism explained above. Sequential determinism resolves the second one. Overall determinism assures that at each state, either no sub-word is recognizable by HA or there is exactly one sub-word that is recognizable and it is recognized by exactly one transition; runs in deterministic automata are uniquely determined.

Lemma 7.4.1 (Unique runs in deterministic HA) Given a deterministic hierarchical automaton HA and a word $w \in symbols^e(HA)^*$. Then each two runs of HA over w starting in the same state are equal. □

The proof in App. C.5, page 224, goes by straightforward induction on w. Before we show how we combine query syntax and semantics, we shortly recall the standard translation of right-linear grammars to basic HA; we need it when proving that dominated product automata accept well-formed queries only.

Definition 7.4.3 (HA for right-linear grammars) Given $G := (N, \Omega, P, S)$ such that G is right-linear. Then the *equivalent hierarchical automaton* $HA(G)$ *for* G is determined by the following components:

$$
\begin{array}{lcl}
\Omega_{HA(G)} & := & \Omega \\
Q_{HA(G)} & := & N \cup \{X\}, X \in N \\
Q^I_{HA(G)} & := & \{S\} \\
Q^F_{HA(G)} & := & \{X\} \\
HA^s_{HA(G)} & := & \emptyset \\
\delta_{HA(G)} & := & \{(A, \mathtt{a}, B) \mid (A \to \mathtt{a}B \in P\} \cup \{(A, \mathtt{a}, X) \mid (A \to \mathtt{a} \in P\}
\end{array}
$$

□

The left-hand part of Fig. 7.5 shows a right-linear grammar that generates the same language as $sep(segment, sep(IPv4Address, G_{aURI}))$. The right-hand part of Fig. 7.5 depicts the HA resulting from the construction in Defn. 7.4.3. For clarity, we have labeled the states with the corresponding non-terminal.

The notions of acceptance and derivability in automata and context-free grammars, respectively, coincide; the following can directly be concluded.

Lemma 7.4.2 ($HA(G)$ is equivalent to G) Given a right-linear grammar G. Then $L(G) = L(HA(G))$. □

The proof can be found in App. C.5, page 224. By separating non-terminals from grammars G we have put a kind of hierarchy on grammars. The production of IP addresses, e.g., is "sourced out" in this way. According to the following theorem, grammar composition is equivalent to the composition of hierarchical automata.

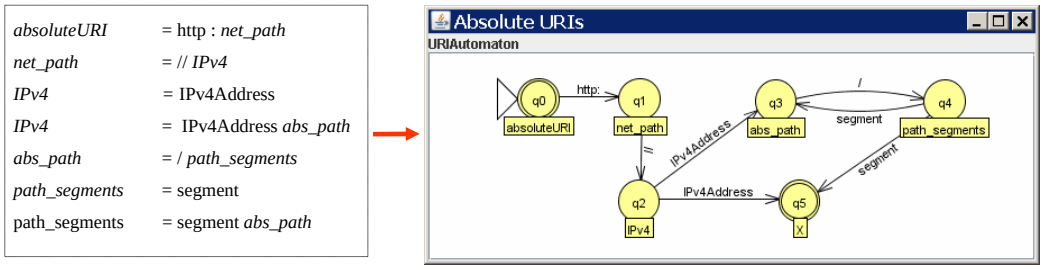


Figure 7.5: Right-linear grammar for absolute URIs and its corresponding FSA

Theorem 7.4.1 (Grammar composition and HA composition are equivalent) Given two right-linear grammars $G := (N, \Omega, P, S)$ and $G' := (N', \Omega', P', S')$ that are composable. Furthermore, let $HA(G)$ and $HA(G')$ be their representing HA according to Defn. 7.4.3. Then $L(G; G') = L(HA(G)[S' \uparrow HA_{G'}])$, where $HA[\mathtt{a} \uparrow HA']$ is defined iff HA' is basic, $HA' \notin HA^s$, and $\mathtt{a} \in HA_\Omega$. In this case, $HA[\mathtt{a} \uparrow HA']$ is determined by the following components:

$$
\begin{array}{lcl}
\Omega_\uparrow & := & HA_\Omega \backslash \{\mathtt{a}\} \\
Q_\uparrow & := & HA_Q \\
Q^I_\uparrow & := & HA_{Q^I} \\
Q^F_\uparrow & := & HA_{Q^F} \\
HA^s_\uparrow & := & \{HA'\} \cup HA^s \\
\delta_\uparrow & := & \{(q, \mathtt{b}, q') \mid (q, \mathtt{b}, q') \in \delta_{HA}, \mathtt{b} \neq \mathtt{a}\} \cup \{(q, HA', q') \mid (q, \mathtt{a}, q') \in \delta_{HA}\}
\end{array}
$$

□

The proof in App. C.5, page 224, constructs runs over $HA(G)[S' \uparrow HA(G')]$ for derivations in $G; G'$ and vice versa. According to this theorem, organizing decomposed grammars into hierarchical automata is well-defined; the resulting automaton accepts exactly the same language that is produced by the re-composed grammar.

Corollary 7.4.1 Given a right-linear grammar $G := (N, \Omega, P, S)$ and a non-terminal $A \in N, A \neq S$. Then there are right-linear grammars $sepG$ and $redG$ with $L(sepG) = L(sep(A, G))$ and $L(redG) = L(red(A, G))$ such that

$$L(G) = L(HA(sepG)[A \uparrow HA(redG)]).$$

□

There, $sepG$ is the result of "re-right-linearizing" $sep(A, G)$; $redG$ results from a language-preserving renaming of non-terminals of $red(A, G)$ such that $sepG$ and $redG$ are composable. The process of "re-right-linearizing" $sep(A, G)$ has been explained before. Applied to our example, this corollary states that our grammar for absolute URIs and the related hierarchical FSA define the same language.

The following construction is closely related to product automata. It facilitates to evaluate *and* generate graph-based queries in an automated way. However, the result automata work more efficiently than standard product automata. Using hierarchical automata turns out to be advantageous here. The *dominating* automaton is *hierarchical*

and carries syntactic elements of graph queries in its edge labels as well as those automata that construct the semantic parts. Related to our example, the automaton HA_{aURI} (cf. Fig. 7.2) is dominating. The automaton for the queried structure is *basic* and carries the path segments *as symbols*. This reduces the number of states and transitions in the product automaton. Also, its states being servers, directories, or documents, carry the query semantics.

Definition 7.4.4 (Dominated product automaton) Given two hierarchical automata

$$SYN := (\Omega_{syn}, Q_{syn}, Q^I_{syn}, Q^F_{syn}, HA^s_{syn}, \delta_{syn})$$
$$SEM := (\Omega_{sem}, Q_{sem}, Q^I_{sem}, Q^F_{sem}, HA^s_{sem}, \delta_{sem})$$

such that SEM is basic. Then the *dominated product* $domProd(SYN, SEM)$ *of* SYN *and* SEM is determined by the following components:

$$\begin{aligned}
\Omega &:= \Omega_{syn} \cup \Omega_{sem} \\
Q &:= \{(q, q') \mid q \in Q_{syn} \ \wedge \ q' \in Q_{sem}\} \\
Q^I &:= \{(q, q') \mid q \in Q^I_{syn} \ \wedge \ q' \in Q^I_{sem}\} \\
Q^F &:= \{(q, q') \mid q \in Q^F_{syn} \ \wedge \ q' \in Q^F_{sem}\} \\
HA^s &:= \emptyset \\
\delta &:= \{(q_s, q'_s), w, (q_t, q'_t) \mid (q_s, a, q_t) \in \delta_{syn}, a \in HA^s_{syn}, (q'_s, w, q'_t) \in \delta_{sem}, w \in L(a)\} \cup \\
&\quad \{(q_s, q'), s, (q_t, q') \mid (q_s, s, q_t) \in \delta_{syn}, s \in \Omega_{syn}, q' \in Q_{sem}\}.
\end{aligned}$$

□

Denoting the automata by SYN and SEM we emphasize that the dominating automaton recognizes *syn*tax, whereas the other one carries *sem*antics in its states. It is easy to see that $domProd(SYN, SEM)$ yields a *basic* HA.

The state set of $domProd(SYN, SEM)$ is given by the Cartesian product of the states of SYN and SEM, respectively. Initial states (q, q') are composed of the initial states of SYN and SEM, respectively. In our example, query evaluation always starts at *Server* objects since we implement absolute URIs. Final states (q, q') are composed of final states of SYN and SEM, respectively.

The transition relation is constructed differently from that of regular product automata. We let both automata run in parallel as long as transition labels of SYN *and* SEM are produced (first disjunction term for δ in Defn. 7.4.4). If, however, symbols are produced that belong to the query language only and, thus, have no correspondence in the queried structure, we stop SEM and proceed in SYN (second disjunction term). We say SYN *dominates* SEM in this case. Due to this procedure, users can neglect purely syntactic material like / when specifying query structures.

Example 7.4.3 (Dominated product automaton)

In Fig. 7.6 we show the link automaton (for absolute links) for the website "Calculation" of the running example; it has been generated fully automatically. However, we have omitted all isolated states. We have annotated the states with labels in order to show their origin. All states are tuples consisting of a state of HA_{aURI} and a state of the query structure. The initial state q_{42}, e.g., is composed of q_0 and the server object (denoted by its IP 137.193.060.082). The transition labeled with "`http` : " indicates that HA_{aURI} dominates; the result state q_{43} still includes the server object

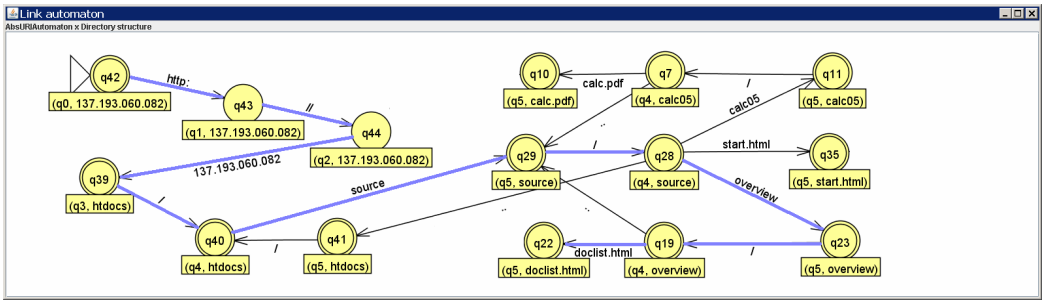


Figure 7.6: Generated link automaton for the website "Calculation"

(no state change in the query structure), but has changed from q_0 to q_1 in HA_{aURI}. The transition labeled with "`137.193.060.082`" from q_{44} to q_{39} shows a state change in both automata.
We have highlighted an accepting run for our sample link

$$\texttt{http://137.193.060.082/source/overview/doclist.html}.$$

It ends in q_{22}, which carries the file "doclist.html" as its semantics.

□

The graphical visualization of the link automaton in Fig. 7.6 directly shows a major benefit of this approach. Apart from evaluating links, it can be used to construct valid links as well. This can be reduced to finding a path / shortest path in a graph. Since our graphs can have cycles, worst case runtime complexity of the best known shortest path algorithms is $O(|V|^2)$. First, we argue that this is an acceptable cost for the achieved benefits. Second, our approach significantly reduces the number of nodes and edges of all automata that have to be processed. In particular, we do not "unfold" or flatten hierarchical automata (those for the query syntax). We rather compute the dominated product directly. In particular, the automaton for the query structure carries full edge labels as *symbols*. The next lemma, applied to our example, states that all accepting runs indeed conform to well-formed URLs. Also, all accepting runs can be reduced to accepting runs in the automaton that represents the query structure.

Lemma 7.4.3 (Languages accepted by dominated product automata) Given a hierarchical automaton SYN and a basic HA SEM. Then the following holds:

(1) $L(domProd(SYN, SEM)) \subseteq L(SYN)$.

(2) If $\Omega_{SEM} \cap \Omega_{SYN} = \emptyset$, then

$$h(s) := \begin{cases} s, & s \in \Omega_{SEM} \\ \epsilon, & s \in \Omega_{SYN} \end{cases}$$

is a language homomorphism from $L(domProd(SYN, SEM))$ to $L(SEM)$.

□

The proof can be found in App. C.5, page 226. Property (2) underlines well-definedness of *domProd*. It holds in our example since none of the edge labels in the

left-hand part of Fig. 7.2 equals any of the symbols "`http` : ", "/" or "//". The next lemma states that this construction applied to deterministic HA produces a deterministic HA as well. This is particularly important if unique query semantics is desired.

Lemma 7.4.4 (Dominated product preserves determinism) Given a hierarchical automaton SYN and a basic HA SEM. Then $domProd(SYN, SEM)$ is deterministic if SYN is deterministic and SEM is branching deterministic. □

The proof is listed in App. C.5, page 226. Since query semantics is defined w.r.t. the result state of a dominated product automaton, the query semantics is uniquely determined if the query automaton is branching deterministic and the automaton for query semantics is deterministic. Notice that determinism is usually desirable for the automaton representing the query language, but not necessarily for the other one. XPath queries, e.g., always link to sets of XML nodes.

7.5 Summary

We have demonstrated that including graph-based queries into our framework is feasible and useful. Since queries usually occur as document content, we have introduced a *specification scheme* for extracting *query structures* from datatype specifications. This results in labeled graphs that carry digital objects as states; states yield query semantics. Query languages are described by context-free grammars, where we restrict ourselves to right-linear grammars; they can be translated into equivalent FSA.

We have introduced *dominated product automata* as a means for combining query syntax and query semantics efficiently. Having translated both, the query grammar and the graph structure, into FSA, the resulting dominated product automaton is generated similarly to regular product automata. It combines both "worlds" using designated nonterminals of the underlying grammar. We have proved that dominated product automata accept well-formed graph queries only. Also, this construction preserves determinism, which is particularly important in our scenario: Web URLs point to exactly one location.

As a benefit of this integration, graph-based document properties can be specified in a more comfortable manner compared to using FOPL formulas. Also, we can evaluate *and* generate queries automatically, which has been demonstrated by automated URL processing; the overall procedure is formally well-founded and meets our claim for high trustworthiness.

Part IV

Case Study

Chapter 8

Case Study — Website Transformation

We apply our approach to automated quality assurance to the website transformation example described in Sect. 2.1, page 16. In particular, we

- use our *functional language* for programming the transformation,
- specify datatypes and *concepts* that are subject to preservation as per Fig. 2.3,
- express the preservation requirements of Fig. 2.3 formally using our *preservation language*, and
- *check* the migration process for adherence to these requirements in an automated way.

Runtime measurements will show that our methods scale to a relevant problem size. For convenience, we have re-listed the informal requirements of Fig. 2.3 in Fig. 8.1.

Preservation requirement (6) (link consistency) will be assured in a fully automated way. We use our *automata-based techniques* of Chap. 7 in order to generate well-formed and valid URLs. There, the case study does not aim at full coverage of the related web-technologies. It is rather reduced to *adequate sample concepts* that are complex enough

(1)	The transformation result matches the *BWeb* format.
(2)	Preserve file names as far as possible.
(3)	Preserve directory names as far as possible.
(4)	Preserve the website's name if possible.
(5)	Preserve the title of the website.
(6)	Preserve link consistency while transforming absolute to relative links.
(7)	Preserve content and structure of the html-files as far as possible.
(8)	Keep the bit-wise content of all non-html files unchanged.
(9)	Preserve the directory and file structure of the source website in both the "html" and "resources" directory.

Figure 8.1: Preservation requirements for case study

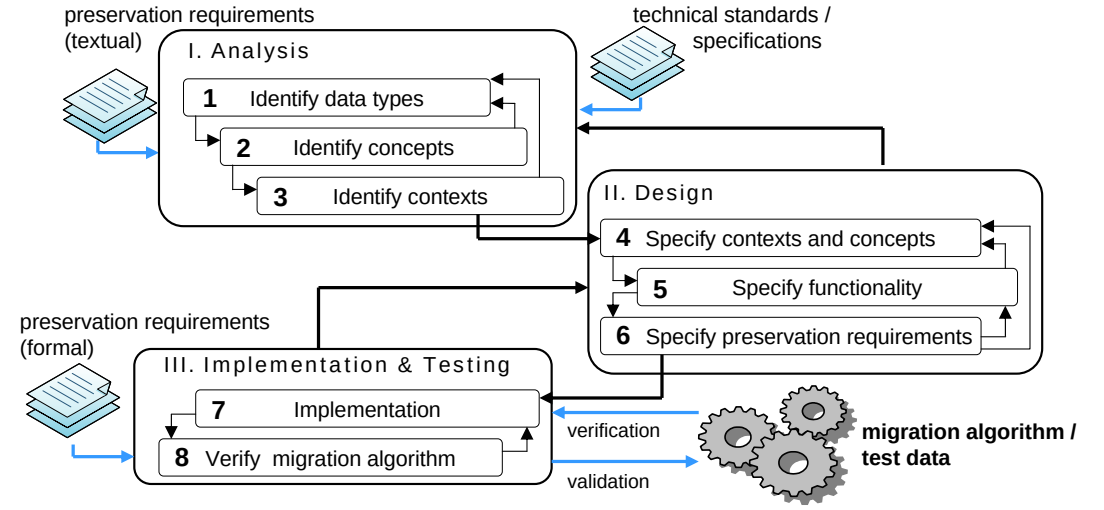


Figure 8.2: Methodology used for case study

for evaluating our method. Another case study dealing with a file format transformation between ODF ([Org06]) and XHTML ([Wor02]) (both annotated with CSS ([Wor07a]) layout information) can be found in [Bor07].

In the following section we describe our methodological approach. It has already been tested in [Bor07] and may be used as a reference. Yet it is not intended to fix a general procedure model. Further case studies will be carried out in future. On this basis we plan to develop best practices and a more profound procedure model.

8.1 Methodology

Fig. 8.2 shows our methodological approach. It is in wide areas reflected by the structure of this thesis. We have run through three phases (*analysis*, *design*, and *implementation & testing*) that comprise a total of eight steps. Knowing that identifying and formulating preservation requirements is a non-trivial process itself, we explicitly state that we require this input for the analysis phase. This is the connecting point of our method to the digital archiving world; we cannot support archives in identifying preservation requirements strategically.

In the analysis phase we have identified relevant data types (step one) as well as relevant concepts and contexts (steps two, three). We do not re-produce the single steps but highlight some points. The preservation requirements in Fig. 8.1 directly suggest the data types `Dir`, `Doc`, `HTMLDoc`, and `Website`; the specification contains terms “directory”, “file”, “html file”, and “website”. Similarly, some concepts can be identified directly. The terms “name”, “content”, “structure”, “format” and “link consistency” occur in different “settings”. The corresponding concepts will be Name, AContent, Contains, EntryPoint, and LinksTo. Recall that concept identification also includes role identification. There, we suggest using meaningful role names. The concept LinksTo, e.g., has been specified between a link source **source**, a link **anchor**, and a link **target**. In all that, existing domain-specific taxonomies or ontologies may serve as an orientation (see Chap. 14).

The different "settings" will be implemented in formal contexts. Requirements (2) to (4), e.g., speak of "directory", "file", and "website" names; this suggests respective contexts for the concept Name. Requirements (7) and (8) deal with the content of html and non-html files, respectively, which suggests corresponding contexts of AContent as well. Also, the last example shows that step one and two are usually carried out iteratively; modeling file content in our example requires for new data types and, hence, for switching back to step one.

After having completed (the first iteration of) phase one, we have started the design phase by specifying concepts and contexts formally. There, additional technical documents were necessary. In particular, there are web standards specifying the syntax and semantics of absolute and relative links. Also, the HTML format is standardized. Of course, this could have been considered in the analysis phase already. We, however, propose the other approach. It keeps the analysis model abstract enough for being discussed with archivists. To be more precise: Does an archivist care about an `<a>` element that implements html links? We consider this to be a matter of technical specification and prefer the more abstract phrase "anchor" that has been used above. As incorporating technical specifications may well induce changes to existing analysis models, we support iteration cycles between phase one and two.

In step five we have specified all functionality that was necessary to implement contexts and concepts. In this case study we have kept contexts as simple as possible and sourced out complex functionality to predicates and functions. This corresponds to a stepwise refinement (top-down), which keeps concept specifications readable and also speeds up concept evaluation; functions and predicates will be *implemented* and *compiled* later on while formulas that occur in concept specifications are evaluated fully formally in our system.

At the end of the design phase we have specified the formal preservation requirements, which required the least effort compared to steps one to five; preservation formulas hide implementation complexity in order to be comprehensible with only minor mathematical knowledge. The "technical" work has to be done in advance. Link consistency is a good example. We have dedicated a whole chapter (Chap. 7) to describing of how to model this property; the related preservation requirement itself, however, is expressed quite easily.

In phase three the formal output of the design phase has to be implemented using our prototype system. This includes implementing the data types as JAVA classes and registering the formal signature that derives from the above specifications. In our case, we also had to implement the migration algorithm itself from scratch. If an existing algorithm is to be checked, it has to be connected to our system. The resulting JAVA implementation, however, is used in step eight to check the underlying transformation process against the preservation requirements.

Notice the co-action between phase three and the migration algorithm (shown by the arrows annotated with "verification" and "validation"). The output of the formalization process above may lack *validity* while properties like soundness can be proved. In contrast, proving formal correctness of complex migration algorithms may be impossible in practice. Hence, both, adapting the specification (backwards arrow from implementation phase to design phase) and adapting the migration algorithm may be sensitive when

our formal system reports violations; we advocate validating specifications and verifying preservation requirements interactively.

8.2 Outline

In the following chapters we present the *results* of the case study. We do *not* recapitulate the single steps shown in Fig. 8.2. The outline is as follows:

(1) Data types for website structures and html content will be introduced in Chap. 9.

(2) In Chap. 10 we introduce the migration algorithm that we have used to migrate websites. It has been implemented using our functional language and incorporates automated link generation.

(3) Concepts are modeled in Chap. 11. There, we will extend the data types of Chap. 9 by additional functionality where necessary.

(4) Formal preservation constraints are provided in Chap. 12. Also, we evaluate runtime performance of our prototype system; we check all preservation requirements for differently sized models.

(5) In Chap. 13 we summarize our results; costs and benefits of our formal quality assurance approach are discussed.

Chapter 9

Modeling Datatypes

We separate two blocks. First, we introduce all data types related to websites and directory structures. Second, we model html content. There, we will not provide the full data type specifications but rather highlight important aspects.

9.1 Modeling Websites, Servers, and Directory Structures

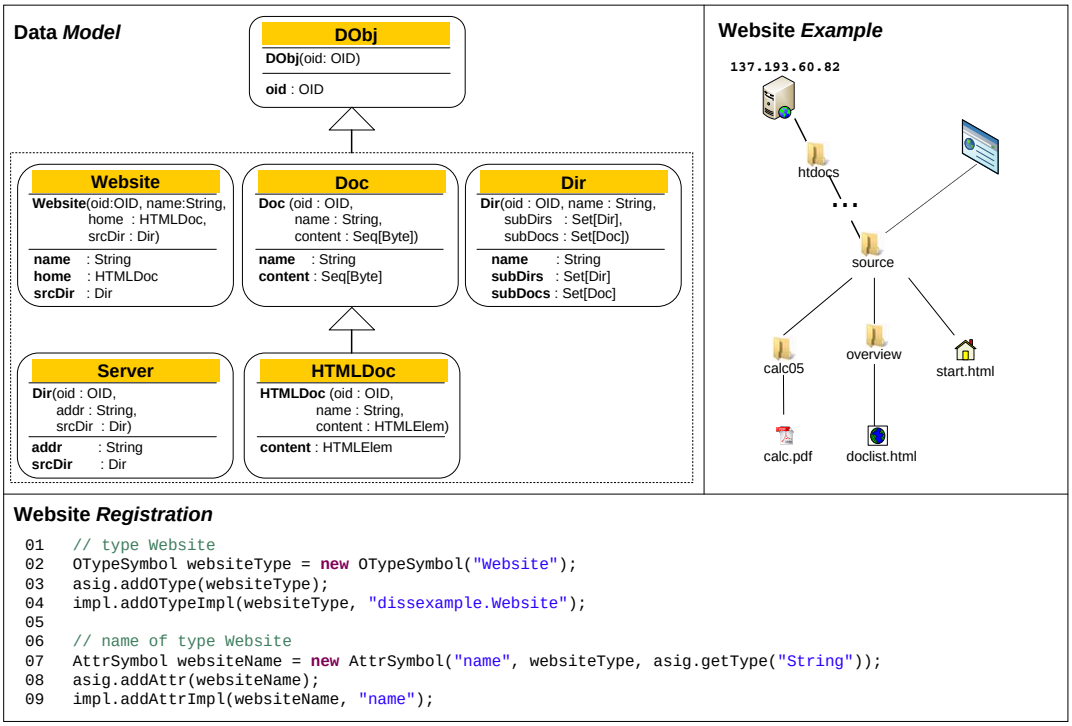
In Fig. 9.1 we provide the website model together with a code snippet that demonstrates datatype registration in our prototype system. Also, we show our example website "Calculation" in the right-hand part of Fig. 9.1 for convenience and reference. The data model in the left-hand part does not contain any functionality so far. It merely shows those attributes[1] that reflect the content of the different data types.

All data types in Fig. 9.1 are object types; they are subtypes of type `DObj` (shown by a grouping box and a UML-like extension arrow). All attributes of super-types are inherited by their sub-types. In Fig. 9.1 we have omitted inherited attributes for brevity.

Type `Website` models websites. Type `Server` represents servers. It does in no way cover real-life web-servers but suffices to model simple web links. In particular, the `addr` attribute carries the web-address of a server. The source directory is stored by the `srcDir` attribute. Since we have used an Apache Web Server for our case study, the source directory of the web-server is "htdocs" (cf. right-hand part of Fig. 9.1).

Directory structures are modeled by type `Dir`. Directories have a *non-empty* name and contain *sets* of sub-directories and sub-documents; we do not further restrict directory names for the sake of simplicity. Type `Doc` covers documents. Again, document names must not be empty. Content (attribute `content`) is represented as a byte sequence. The basic types `Byte` and `Seq` are standard and are not shown for brevity. The so-defined type `Doc` covers arbitrary *static* web-documents. Streaming (like RSS-feeds) cannot be represented. Type `HTMLDoc` models html documents. It is a sub-type of `Doc` and, hence, inherits all respective attributes. It, however, overrides the `content` attribute. We will provide the data model for html content in the next section. Notice that, sub-directories and sub-documents of given directories must have distinct names; file paths may be ambiguous otherwise.

[1] In the context of ADT specifications, attributes are rather called *selectors*. In the DA community, however, the term *attribute* is more common; we adapt the latter.



Website *Registration*

```
01  // type Website
02  OTypeSymbol websiteType = new OTypeSymbol("Website");
03  asig.addOType(websiteType);
04  impl.addOTypeImpl(websiteType, "dissexample.Website");
05
06  // name of type Website
07  AttrSymbol websiteName = new AttrSymbol("name", websiteType, asig.getType("String"));
08  asig.addAttr(websiteName);
09  impl.addAttrImpl(websiteName, "name");
```

Figure 9.1: Websites, servers, and directory structures

To generate a feeling for the required implementation work, the code snippet in the bottom part of Fig. 9.1 shows how type `Website` and its `name` attribute are registered in our prototype system. In line two, we define a new object type symbol for `Website`; class `OTypeSymbol` represents object types. After that, the so-defined type symbol is registered in the current signature `asig`; `asig` is an instance of the class `ArchSignature`, which represents signatures. `Website` is implemented in the JAVA class `dissexample.Website`. In line four, this class is registered as `Website`'s interpretation (semantics). There, `impl` is an instance of the class `Implementation`; `Implementation` administrates symbol interpretations in our system.

The `name` attribute is registered in line seven following the same scheme. A symbol is declared (of class `AttrSymbol`) and added to both the signature and implementation. In particular, line nine registers the JAVA-method `name` of class `Website` as interpretation for the `name` attribute.

9.2 Modeling Html Content

We represent html documents as DOM - trees (*Document Object Model*, [Wor04a]). The elements shown in the left-hand part of Fig. 9.2 directly reflect those parts of the DOM specification that we support in our case study. In order to have an example, we have listed the content of the file "start.html" in the right-hand part of Fig. 9.2. The data model represents static html content only. Dynamic aspects like html forms and related events cannot be represented. Notice that we use *basic* types in the first instance. Type `HTMLContElem` is the super-type of all html content types. It introduces the attribute

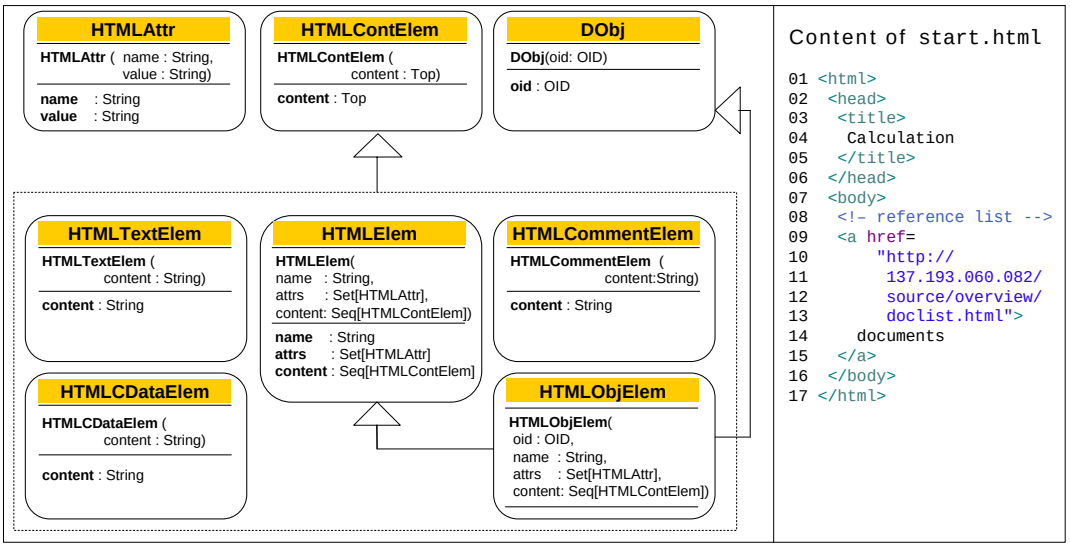


Figure 9.2: Content model for html documents

`content` : `HTMLContElem` → `Top` representing general content (recall that `Top` is the super-type of all types). The sub-types of `HTMLContElem` refine this attribute. Text elements (`HTMLTextElem`), comment elements (`HTMLCommentElem`), and CDATA elements (`CDataElem`) have textual content; the `content` attribute has return type `String`. Type `HTMLElem` is recursive and models complex content. It can have attributes (`attrs`) as well as further sub-elements (`content`). There, we require all attributes to have mutually distinct names. Html attributes are modeled by type `HTMLAttr`. They have a name (`name`) and (`value`) (cf. `href`-attribute in the right-hand part).

Apart from these basic types we introduce an *object* type `HTMLObjElem`, which is a sub-type of `HTMLElem`. Recall that we trace histories of digital objects only. When checking preservation of link-consistency we will have to trace link anchors. For the sake of simplicity, we model links in `<a>` elements only. Tracing link anchors, thus, corresponds to tracing `<a>` elements. Tracing other than `<a>` elements, however, is not necessary. To keep the number of objects in the system to a minimum, we model html content by basic elements and only keep `<a>` elements as digital objects. We propose this technique as a general design pattern whenever data structures have to be traced only partly; our support for multiple inheritance is vital in this setting. In the next section we provide the formal signature that derives from the data types introduced so far.

9.3 Formal Signature

Apart from the static parts of the basic DA (cf. Tab. 3.4, page 42), the current (static) signature consists of the components shown in Tab. 9.1. We have listed it for later reference. The output is generated by our system and particularly shows the benefits of overloading. Equal aspects (like directory, website, or document names) can be implemented for different types by equally named functions. This concludes the introduction of the basic data model. In the next chapter we introduce the underlying transformation, which has been implemented using our functional language.

Table 9.1: *Basic signature for case study*

Types and $<$:		
Dir	$<$	{Top, Dir, DObj}
Doc	$<$	{Top, Doc, DObj}
HTMLDoc	$<$	{Top, Doc, HTMLDoc, DObj}
Website	$<$	{Top, Website, DObj}
Server	$<$	{Top, Server, DObj}
HTMLContElem	$<$	{Top, HTMLContElem}
HTMLElem	$<$	{HTMLElem, Top, HTMLContElem}
HTMLObjElem	$<$	{HTMLElem, Top, HTMLContElem, HTMLObjElem, DObj}
HTMLCommentElem	$<$	{Top, HTMLContElem, HTMLCommentElem}
HTMLCDataElem	$<$	{Top, HTMLContElem, HTMLCDataElem}
HTMLTextElem	$<$	{Top, HTMLContElem, HTMLTextElem}
HTMLAttr	$<$	{Top, HTMLAttr}
Byte	$<$	{Byte, Top}
String	$<$	{Top, String}

Functions:	
name : Dir → String	name of a Dir
subDirs : Dir → Set[Dir]	sub-directories of a Dir
subDocs : Dir → Set[Doc]	sub-documents of a Dir
content : Doc → Seq[Byte]	content of a Doc
name : Doc → String	name of a Doc
content : HTMLDoc → HTMLElem	content of an HTMLDoc
home : Website → HTMLDoc	home of a Website
name : Website → String	name of a Website
srcDir : Website → Dir	source directory of a Website
addr : Server → String	web-address of a Server
srcDir : Server → Dir	source directory of a Server
content : HTMLContElem → Top	content of an HTMLContElem
name : HTMLElem → String	name of an HTMLElem
attrs : HTMLElem → Set[HTMLAttr]	attributes of an HTMLElem
content : HTMLElem → Seq[HTMLContElem]	content of an HTMLElem
content : HTMLCommentElem → String	content of an HTMLCommentElem
content : HTMLCDataElem → String	content of an HTMLCDataElem
content : HTMLTextElem → String	content of an HTMLTextElem
name : HTMLAttr → String	name of an HTMLAttr
value : HTMLAttr → String	value of an HTMLAttr

Constructors:
Dir : OID × String × Set[Dir] × Set[Doc] → Dir
Doc : OID × String × Seq[Byte] → Doc
HTMLDoc : OID × String × HTMLElem → HTMLDoc
Website : OID × String × HTMLDoc × Dir → Website
Server : OID × String × Dir → Server
HTMLContElem : Top → HTMLContElem
HTMLElem : String × Set[HTMLAttr] × Seq[HTMLContElem] → HTMLElem
HTMLObjElem : OID × String × Set[HTMLAttr] × Seq[HTMLContElem] → HTMLObjElem
HTMLCommentElem : String → HTMLCommentElem
HTMLCDataElem : String → HTMLCDataElem
HTMLTextElem : String → HTMLTextElem
HTMLAttr : String × String → HTMLAttr

Chapter 10

Implementing the Migration

Recall that our transformation algorithm is expected to satisfy the preservation requirements in Fig. 8.1. To achieve this goal, our algorithm works in three steps:

(1) Structural transformation: Transform the source directory and file structure such that it conforms to *BWeb*; do not alter html content.

(2) Adaptation: Parse the title of the website and possibly adapt the names of the source directory of the target website and of the target website itself (cf. constraints of the *BWeb* format and preservation requirement (1)).

(3) Content migration: Migrate all html content; use link automata to evaluate links in the source website and to generate suitable relative links for the target website.

Step three includes "garbage collection" as well. We, however, keep the source website.

We deal with these steps separately in the following sections. There, we do not introduce the full algorithm; we highlight important aspects only. In particular, the algorithm is to demonstrate how automated link construction can be incorporated.

10.1 Structural Transformation

The result of this transformation step is visualized in Fig. 10.1. The output is twofold. First, we generate a target website that conforms to the *BWeb* format and contains all content of the source website. Also, the input website is maintained. In order to distinguish source and target, we have underlined all components of the target website. Second, we generate a transformation history ("Traces" box on the right-hand side in Fig. 10.1). This transformation history is used in step three for generating relative links with appropriate source and target files; recall that we have to transfer links between objects in the source structure to their transformation results in the target structure.

Step one meets requirements (2) to (5), (7) to (9), and, in large parts, requirement (1) (see Fig. 8.1, page 122). The title of the target website does not yet necessarily match the name of the website; this will be assured in step two. Also, html links are not yet adapted; we simply copy the original html content. The html files of the target website, thus, still point to the original locations (dotted arrows).

The main function for step one is typed as follows:

$$\texttt{migAWebToBWeb} : \texttt{Website} \rightarrow \texttt{Tuple}[\texttt{Website}, \texttt{Set}[\texttt{Tuple}[\texttt{DObj}, \texttt{DObj}]]].$$

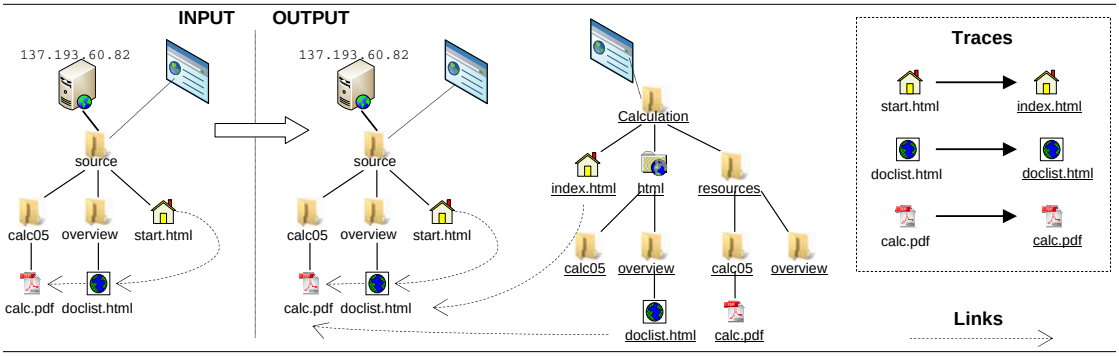


Figure 10.1: Result of transformation step one

It takes a website w and returns a tuple containing the transformation target of w and the transformation history. Histories are modeled by sets containing mappings from source files to their transformation results.

In the body

```
migAWebToBWeb(w) =
    let  new_home  =  migHTMLDoc(home(w), w)
         new_src   =  migSrcDir(srcDir(w), w, new_home)
    in   Tuple(transform(w ↦ (Website, name(w), new_home, fst(new_src))), snd(new_src))
```

the target website receives a new source directory and home page, which are computed by `migHTMLDoc` and `migSrcDir`, respectively. The function `migHTMLDoc` has two input parameters — the html document d that is to be transformed and a website w. If the html document is the home page of the website, the resulting html document is named "`index.html`", as required by the *BWeb* format. Otherwise, the name of the source document is preserved.

The function `migSrcDir` is given by:

```
migSrcDir(d, w, new_home) =
    let  rec_dir       =  {migDirHTML(x, w) | x ← subDirs(d), True}
         htmlSubDirs   =  {fst(x) | x ← rec_dir, True}
         rec_doc       =  {let r = migHTMLDoc(cast(x, HTMLDoc), w)
                           in Tuple(r, {Tuple(cast(x, HTMLDoc), r)}_s)
                              | x ← subDocs(d), and(inst(x, HTMLDoc), not(x = home(w)))}
         htmlSubDocs   =  {fst(x) | x ← rec_doc, True}
         htmlDir       =  create(Dir, "html", htmlSubDirs, htmlSubDocs)
         ...           =  ...
         resDir        =  create(Dir, "resources", resSubDirs, resSubDocs)
         all_trans     =  joinTrans(rec_dir) ∪ joinTrans(rec_doc)∪
                          joinTrans(rec_dir2) ∪ joinTrans(rec_doc2)
    in   Tuple(transform(d ↦ (Dir, name(w), {htmlDir, resDir}, {new_home}_s)), all_trans)
```

It has three input parameters — the directory d that is to be transformed, a website w, and the new home page *new_home* of w.

According to the *BWeb* format, a website's source directory must exactly contain a directory "`html`" for html content, a directory "`resources`" for non-html content, and the website's home page named "`index.html`". In the `let` part, `migSrcDir`, thus, executes the following steps:

- It uses `migDirHTML` to clone the sub-directory structure of d for the "`html`" directory. The result type of `migDirHTML` is `Tuple`[`Dir`, `Set`[`Tuple`[`DObj`, `DObj`]]]; the result carries the transformation target and a mapping reflecting all executed transformations. The overall result is stored in *rec_dir*. Also, `migDirHTML` assures that all included directories contain html content only.
- It stores the new sub-directories of "html" in *htmlSubDirs*. There, `fst` and `snd` return the first and second element of a tuple, respectively.
- It uses `migHTMLDoc` to transform the html documents of d. The result type of `migHTMLDoc` is `HTMLDoc`.
- It creates the "`html`" directory *htmlDir* and attaches the just-created sub-directory structure.
- It creates the "`resources`" directory *resDir*; the sub-directory structure is generated in the parameters *rec_dir*2, *resSubDirs*, *rec_doc*2, and *resSubDocs* in the same way it has been created for "html" (details are omitted, which is indicated by dots). Instead of `migDirHTML` and `migHTMLDoc`, however, `migDirResources` and `transform` are used. There, `migDirResources` assures that the result contains non-html content only.
- It uses `joinTrans` to join *the histories* of all executed transformations; `joinTrans` has result type `Set`[`Tuple`[`DObj`, `DObj`]].

Finally, `migSrcDir` transforms the website w and attaches the respective components in the constructor call. In this way, requirements (1) and (3) of the *BWeb* format are satisfied.

10.2 Adaptation

In order to fully assure that the target website is in *BWeb* format (requirement (1)), we potentially have to adapt the website's name. We use

```
adaptWebName : Website → Website
adaptWebName(w) =
   let title = parseTitle(0, home(w))
   in if or(title = {}, name(w) ∈ title) then w
      else let n = rep(title)
               d = create(Dir(n, subDirs(srcDir(w)), subDocs(srcDir(w))))
           in transform(w ↦ (Website, n, home(w), d))
```

to check whether the home page of the input website w has a title (function `parseTitle`); if it does not, w remains unchanged. The same is true if the name of w already matches the title. Otherwise, w and its source directory are transformed appropriately. There, a string n in *title* is chosen non-deterministically. The implementation of `parseTitle` is not provided for brevity; the specification, however, can be found in Fig. 11.2, page 139.

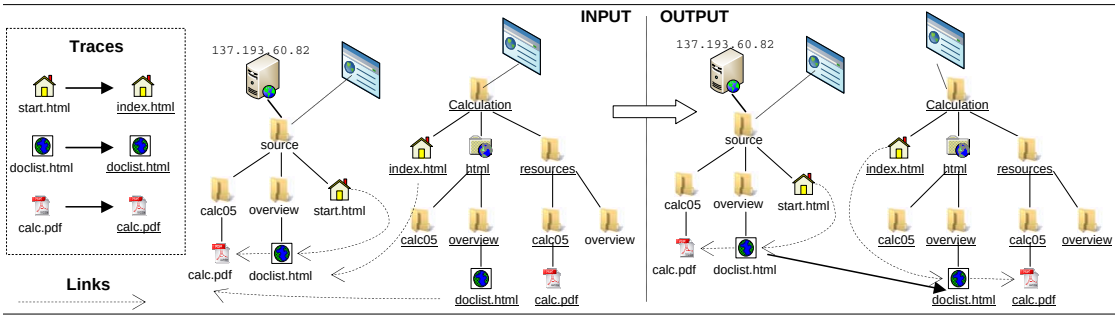


Figure 10.2: Result of transformation step three

10.3 Content Migration

Step three migrates html contents. It is the most challenging one as it includes link generation and "garbage collection"; intermediate structures have to be deleted. In the following we, however, factor out object deletion and focus on link generation. In Fig. 10.2 we show the effect of step three. The input comprises the output of step two and the transformation history generated in step one ("Traces" box). All html contents is to be cloned except for links. They must be adapted to the new structure. The output is shown on the right-hand side. Links (dotted arrows) do not longer point to locations in the source website but point to their corresponding transformation results in the target website. The latter are extracted from the transformation history. The target model in the right-hand part of Fig. 10.2 includes the trace for "doclist.html" in order to visualize how the link from "index.html" is adapted.

Step three is started by calling

$$\texttt{migWebHTMLContent} \ : \ \texttt{Website} \times \texttt{Set}[\texttt{Tuple}[\texttt{DObj}, \texttt{DObj}]] \rightarrow \texttt{Website}.$$

It gets the source website w and the transformation history of step one as input. In the body

```
migAWebHTMLContent(w, trans) =
    let  new_home  =  migHTMLContent(home(w), trans, srcDir(w))
         new_src   =  migSrcDirContent(srcDir(w), trans, new_home)
         result    =  transform(w ↦ (Website, name(w), new_home, new_src))
         ...       ...  ...
    in   result
```

`migAWebHTMLContent` migrates the content of the home page of w using `migHTMLContent` and then migrates all html content recursively that is contained in its source directory (`migSrcDirContent`). The transformation history is passed through to both function calls as it is used to generate new links appropriately. The dots indicate that we have skipped some function calls; we delete intermediate object structures there.

Both, `migSrcDirContent` and `migHTMLContent` basically delegate their calls to

$$\texttt{migHTMLElem} \ : \ \texttt{HTMLContElem} \times \texttt{Set}[\texttt{Tuple}[\texttt{DObj}, \texttt{DObj}]] \times \texttt{Dir} \rightarrow \texttt{HTMLContElem}$$

which migrates html elements. In the following we list the function body in order to demonstrate how we use link automata to generate valid links.

```
migHTMLElem(e, trans, pDoc, pDir) =
    if    inst_HTMLElem(e)
    then  let  ee           =  cast(e, HTMLElem)
               new_content  =  migHTMLElemSeq(content(ee), trans, pDoc, pDir)
          in   if name(ee) = "a" then
                  let nhattrs = {a | a ← attrs(ee), name(a) ≠ "href"}
                      hattrs = {genAttrForLink(ee, a, trans, pDoc, pDir)
                                    | a ← attrs(ee), name(a) = "href"}
                      new_attrs = nhattrs ∪ hattrs
                  in if inst_HTMLObjElem(ee) then
                        transform(ee ↦ HTMLObjElem, name(ee), new_attrs, new_content)
                     else HTMLElem(name(ee), new_attrs, new_content)
               else
                  if inst_HTMLObjElem(ee) then
                     transform(ee ↦ HTMLObjElem, name(ee), attrs(ee), new_content)
                  else HTMLElem(name(ee), attrs(ee), new_content)
    else  e
```

The parameters *pDoc* and *pDir* are the parent html document of the `HTMLElem` e and the parent directory of *pDoc*, respectively. Both are needed to generate relative links. If e is a basic content element (e.g., of type `HTMLTextElem`), it is returned directly. Otherwise, e is casted to `HTMLElem` and stored in ee for convenience. Also, the new content of e is computed recursively; `migHTMLElemSeq` simply calls `migHTMLElem` for all corresponding elements in `content`(e).

If ee is no `<a>` element, we check whether ee is of type `HTMLObjElem`. If true, it is transformed and receives the new content. Otherwise, a new `HTMLElem` (basic type) is returned; it has the new content as well.

If ee is an `<a>` element, we generate a set *new_attrs* that contains the attributes for the transformed `<a>` element. There, *hattrs* contains all new `href` attributes whereas *nhattrs* contains all other attributes. Recall that *hattrs* is empty or a singleton. In *hattrs* we use `genAttrForLink` to generate new links as follows:

```
genAttrForLink(e, a, trans, pDoc, pDir) =
    let      trg = linkTargets(pDoc, e)
    in if    ¬empty(trg)
      then  rep({HTMLAttr(  "href",
                            findWord(RelURIAutomaton(existDObj), pDir, snd(tr)))
                    | tr ← trans, fst(tr) ∈ trg})
      else  a
```

The function `linkTargets` returns all link targets that the `<a>` element e points to. Both, absolute and relative linking is considered. Therefore, `linkTargets` needs an explicit starting point in the first parameter. In our case, this is *pDoc* (parent document of e). If the second parameter contains no valid URI, `linkTargets` returns the empty set. Otherwise, the result contains exactly one element. It is used to create a new attribute. There, the transformation history *trans* is used to determine the transformation results of those objects the original link has pointed to (`fst`(tr) $\in$ *trg*). Since *trg* is a singleton and documents have been transformed only once, at most one resulting new `href` attribute is generated. There, `snd`(tr) carries the transformation result of `fst`(tr) such that `findWord` generates a word leading from *pDir* to the transformation result of

the original link target. Recall that *pDir* is the parent directory of *pDoc*; the relative link is generated between *pDoc* and `snd`(tr). There, `RelURIAutomaton` is instantiated w.r.t. `existDObj` such that the generated link is indeed valid in the current system state.

Among others, the functions used in step three show two things: First, content migration is not trivial in our example; transformations had to be traced "by hand" and several case distinctions have to be made. Indeed, our formal constraints have helped in developing this transformation and in correcting initial functional errors.

Second, our automata-based techniques smoothly integrate into our framework. Automata are regular datatypes. Their functions hide complexity and can be used in a straightforward way. This concludes our explanations on the migration algorithm. In the next chapter we proceed with concept specifications.

Chapter 11

Specifying Concepts

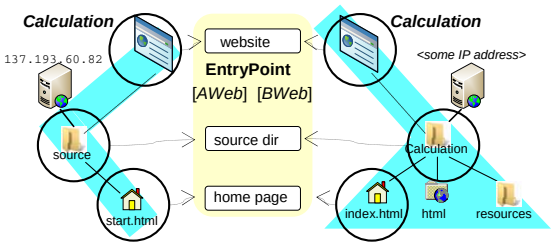
Having implemented the underlying transformation algorithm, we now have to specify those properties that are to be preserved according to the requirements in Fig. 8.1. Tab. 11.1 shows those concepts and contexts that have been extracted from these informal requirements. There, we relate requirements (column one) and general terms that occur in different variants in their formulations (columns two and three) to the corresponding concept, roles, and implementing context (columns four to six). Details of the respective specifications can be found in the sections shown in the last column.

The concepts EntryPoint and Name cover website formats and names, respectively. Link consistency is modeled by the concept LinksTo. There, we distinguish absolute and relative links; both variants significantly differ on the implementation level. Links are identified w.r.t. a link source, a link anchor, and a link target, where the contexts *AbsURI* and *RelURI* implement absolute and relative links, respectively.

The concept AContent covers the term "content". In our example, it will be used in a functional preservation constraint in order to compare the content of files (context

Table 11.1: *Identified concepts and contexts*

Req.	Term	Description	Concept	Roles	Context	Details
(1)	format	*AWeb* format *BWeb* format	EntryPoint	**website**, **sourcedir**, **homepage**	*AWeb* *BWeb*	Sect. 11.1
(2)	name	file name	Name	**entity**	*DocN*	—
(3)		directory name			*DirN*	
(4)		website name			*WebN*	
(6)	link consistency	absolute links	LinksTo	**source**, **anchor**,	*AbsURI*	Sect. 11.5
(6)		relative links		**target**	*RelURI*	
(5),(7)	content	content of html files	AContent	**container**	*HtmlDocC*	Sect. 11.2
(8)		content of non-html files			*DocC*	
(9)		content of directories			*DirC*	
(7)	structure	html file contains html elem.	Contains	**parent**, **child**	*CHtmlElem*	Sect. 11.3
(7)		html elem. contains html elem.			*CElemElem*	
(9)		direct directory containment			*CDir*	
(9)		direct file containment			*CDoc*	
(9)		rec. directory containment			*CDirRec*	
(9)		rec. file containment			*CDocRec*	
(9)		source directory in *BWeb* contains directory in "html"			*CDirHtml*	
(9)		source directory in *BWeb* contains dir. in "resources"			*CDirRes*	
(7)	structure	html elements are neighbors	Neighbor	**left**, **right**	*NHtmlElem*	Sect. 11.4



Specification for *validAWeb*, *validBWeb*

$\forall w : \texttt{Website} \bullet validAWeb(w) \Leftrightarrow$
$\quad \exists d : \texttt{HTMLDoc} \bullet containsDoc(\texttt{srcDir}(w), d)$
$\forall w : \texttt{Website} \bullet validBWeb(w) \Leftrightarrow$
$\quad \texttt{name}(w) = \texttt{name}(\texttt{srcDir}(w)) \wedge$
$\quad \texttt{home}(w) \in \texttt{subDocs}(\texttt{srcDir}(w)) \wedge$
$\quad \texttt{parseBWeb}(0, \texttt{srcDir}(w), \texttt{name}(w)) = \texttt{True}$

$\forall d : \texttt{Dir}, d' : \texttt{Doc} \bullet containsDoc(d, d') \Leftrightarrow$
$\quad d' \in \texttt{subDocs}(d) \vee$
$\quad \exists d'' : \texttt{Dir} \bullet d'' \in \texttt{subDirs}(d) \wedge containsDoc(d'', d')$

concept EntryPoint =
begin
Interface:
$\quad \langle \textbf{website} : \alpha_w, \textbf{sourcedir} : \alpha_d,$
$\quad \textbf{homepage} : \alpha_h \rangle$
$\quad \alpha_w < \texttt{Website} \wedge \alpha_d < \texttt{Dir} \wedge$
$\quad \alpha_h < \texttt{HTMLDoc}$
Context *AWeb*:
ι_A $\quad validAWeb(website) \wedge$
$\quad sourcedir = \texttt{srcDir}(website) \wedge$
$\quad homepage = \texttt{home}(website)$
Context *BWeb*:
ι_B $\quad validBWeb(website) \wedge$
$\quad sourcedir = \texttt{srcDir}(website) \wedge$
$\quad homepage = \texttt{home}(website)$
end EntryPoint;

Figure 11.1: Specification of concept EntryPoint and predicate *containsDoc*

DocC), html-files (*HtmlDocC*), and directories (*DirC*) before and after a transformation. For this purpose, we will partly *abstract* from the real content, which justifies the suffix A in the concept name.

Finally, we have to deal with "structure" — html files have structured content and websites contain directory structures. In the first instance, we interpret structures as a containment relation. Hence, the corresponding concept is called Contains and uses roles **parent** and **child**. The structure of html content, however, adds an additional dimension; the order of the html elements is important. This is covered by the concept Neighbor. It has roles **left** and **right** and is implemented for html elements only (*NHtmlElem*). The concept Contains has several additional implementations that are described briefly in Tab. 11.1.

In the following sections we provide details. There, we omit Name for brevity; it is simple and has already briefly been introduced before.

11.1 The Concept EntryPoint

This concept has already been introduced in Chap. 4. For convenience, we have listed the specification and a visualization again in Fig. 11.1. The two contexts are implemented quite similarly but use different validity checks. In context *AWeb* we check for conformance to the *AWeb* format. This is done by the predicate *validAWeb*, the specification of which is shown in the lower left-hand part of Fig. 11.1. It simply requires the source directory of w to contain an html file. There, *containsDoc* models recursive file containment and belongs to type `Dir`. *BWeb* conformity is implemented by *validBWeb*. It delegates the check to `parseBWeb`.

Before we introduce the full specification, we recall the format description for *BWeb*:

(1) The source directory of the website exactly contains two sub-directories "html" and

"resources", respectively, and a file named "index.html", which is the welcome page of the website.

(2) The names of the source directory and the website are equal. If the website's welcome page has a `<title>` element with non-empty textual content, both names equal the content of this element.

(3) All html-files except "index.html" must reside in the directory "html".

(4) The directory "resources" contains all non-html-files.

This declarative formulation can be translated into suitable FOPL formulas with little effort. Yet, we use a parser, i.e., a *function*, that will be specified *operationally* later on. This considerably speeds up the process of checking *BWeb* conformity on the one hand, but introduces a proof obligation on the other hand. Does successful parsing really indicate *BWeb* conformity? Does parsing succeed for all websites conforming to *BWeb*? Before we can prove these properties, we have to translate the above textual format description into a formal one. The result is provided in the following definition.

Definition 11.1.1 (*BWeb* format and *containsElem*) Given a website w. Then w *conforms to BWeb* iff the following holds:

$$
\begin{array}{ll}
(1) & \texttt{size}(\texttt{subDirs}(\texttt{srcDir}(w))) = 2 \wedge \\
& \exists d' : \texttt{Dir} \bullet d' \in \texttt{subDirs}(\texttt{srcDir}(w)) \wedge \texttt{name}(d') = \text{“}\texttt{html}\text{“} \wedge \\
& \exists d' : \texttt{Dir} \bullet d' \in \texttt{subDirs}(\texttt{srcDir}(w)) \wedge \texttt{name}(d') = \text{“}\texttt{resources}\text{“} \wedge \\
& \texttt{size}(\texttt{subDocs}(\texttt{srcDir}(w))) = 1 \wedge \\
& \texttt{name}(\texttt{home}(w)) = \text{“}\texttt{index.html}\text{“} \wedge \texttt{home}(w) \in \texttt{subDocs}(\texttt{srcDir}(w)) \\
(2) & \texttt{name}(w) = \texttt{name}(\texttt{srcDir}(w)) \wedge \\
& \forall e : \texttt{HTMLElem} \bullet (\ \mathit{containsElem}(\texttt{content}(\texttt{home}(w)), e) \Rightarrow \texttt{name}(e) \neq \text{“}\texttt{title}\text{“}) \vee \\
& \exists e : \texttt{HTMLElem} \bullet (\ \mathit{containsElem}(\texttt{content}(\texttt{home}(w)), e) \wedge \texttt{name}(e) = \text{“}\texttt{title}\text{“} \wedge \\
& \qquad (\forall e' : \texttt{HTMLTextElem} \bullet \ (\mathit{containsElem}(e, e') \Rightarrow \texttt{content}(e') = \text{“”})) \vee \\
& \qquad (\exists e' : \texttt{HTMLTextElem} \bullet \ (\mathit{containsElem}(e, e') \wedge \texttt{content}(e') \neq \text{“”} \wedge \\
& \qquad\qquad \texttt{content}(e') = \texttt{name}(w))) \ ) \\
(3) & \forall d : \texttt{Dir}, d' : \texttt{HTMLDoc} \bullet (\ (d \in \texttt{subDirs}(\texttt{srcDir}(w)) \wedge \mathit{containsDoc}(d, d')) \Rightarrow \texttt{name}(d) = \text{“}\texttt{html}\text{“} \ ) \\
(4) & \forall d : \texttt{Dir}, d' : \texttt{Doc} \bullet (\\
& \quad (d \in \texttt{subDirs}(\texttt{srcDir}(w)) \wedge \mathit{containsDoc}(d, d') \wedge \neg \texttt{inst}_{\texttt{HTMLDoc}}(d')) \Rightarrow \texttt{name}(d) = \text{“}\texttt{resources}\text{“} \ )
\end{array}
$$

There, the predicate *containsElem* reflects recursive containment of an *HTMLContElem* and is defined by

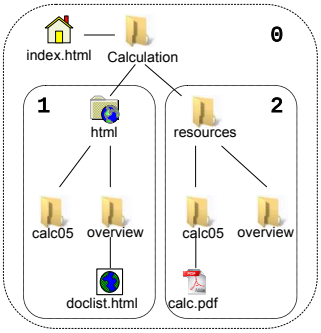
$$
\begin{array}{l}
\forall e : \texttt{HTMLElem}, e' : \texttt{HTMLContElem} \bullet \\
\quad \mathit{containsElem}(e, e') \Leftrightarrow \\
\qquad e' \in \texttt{content}(e) \ \vee (\exists e'' : \texttt{HTMLElem} \bullet \ e'' \in \texttt{content}(e) \wedge \mathit{containsElem}(e'', e'))
\end{array}
$$

□

Formulas (1), (3), and (4) are straightforward realizations of requirements (1), (3), and (4), respectively. In formula (2) we introduce a predicate *containsElem* : `HTMLElem`× `HTMLContElem`, which belongs to type `HTMLElem` and models recursive html element containment. Formula (2) is satisfied if the welcome page of w contains no `<title>` element (line two) or it contains a `<title>` element (line three) that either has only empty textual content (line four) or contains an `HTMLTextElem` the content of which equals the website's name. This implements requirement (2) above. Yet we assume that website titles are permissible directory names for the sake of simplicity. Also, we define the website's title to be the content of *one* text element that occurs as sub-element of a

Specification for `parseBWeb`

$\forall d : \texttt{Dir}, i : \texttt{Nat}, n : \texttt{String} \bullet$
$\texttt{parseBWeb}(0, d, n) = \texttt{True} \Leftrightarrow$
$\quad \texttt{size}(\texttt{subDirs}(d) = 2) \wedge$
$\quad \exists d' : \texttt{Dir} \bullet d' \in \texttt{subDirs}(d) \wedge \texttt{name}(d') = \text{“}\texttt{html}\text{“} \wedge$
$\qquad \texttt{parseBWeb}(1, d', n) = \texttt{True} \wedge$
$\quad \exists d' : \texttt{Dir} \bullet d' \in \texttt{subDirs}(d) \wedge \texttt{name}(d') = \text{“}\texttt{resources}\text{“} \wedge$
$\qquad \texttt{parseBWeb}(2, d', n) = \texttt{True} \wedge$
$\quad \texttt{size}(\texttt{subDocs}(d) = 1) \wedge$
$\quad \exists d' : \texttt{HTMLDoc} \bullet d' \in \texttt{subDocs}(d) \wedge \texttt{name}(d') = \text{“}\texttt{index.html}\text{“} \wedge$
$\qquad (\texttt{parseTitle}(0, \texttt{content}(d')) = \{\} \vee$
$\qquad n \in \texttt{parseTitle}(0, \texttt{content}(d'))) \wedge$
$\texttt{parseBWeb}(1, d, n) = \texttt{True} \Leftrightarrow$
$\quad (\forall d' : \texttt{Dir} \bullet d' \in \texttt{subDirs}(d) \Rightarrow \texttt{parseBWeb}(1, d', n) = \texttt{True}) \wedge$
$\quad (\forall d' : \texttt{Doc} \bullet d' \in \texttt{subDocs}(d) \Rightarrow \texttt{inst}_{\texttt{HTMLDoc}}(d')) \wedge$
$\texttt{parseBWeb}(2, d, n) = \texttt{True} \Leftrightarrow$
$\quad (\forall d' : \texttt{Dir} \bullet d' \in sDirs \Rightarrow \texttt{parseBWeb}(2, d, n) = \texttt{True}) \wedge$
$\quad (\forall d' : \texttt{Doc} \bullet d' \in \texttt{subDocs}(d) \Rightarrow \neg\texttt{inst}_{\texttt{HTMLDoc}}(d')) \wedge$
$i > 2 \Rightarrow \texttt{parseBWeb}(i, d, n) = \texttt{False}$



Specification for `parseTitle`

$\forall e : \texttt{HTMLElem}, i : \texttt{Nat}, n : \texttt{String} \bullet$
$n \in \texttt{parseTitle}(0, e) \Leftrightarrow$
$\quad (\texttt{name}(e) = \text{“}\texttt{title}\text{“} \wedge n \in \texttt{parseTitle}(1, e)) \vee$
$\quad (\exists e' : \texttt{HTMLElem} \bullet e' \in \texttt{content}(e) \wedge n \in \texttt{parseTitle}(0, e')) \wedge$
$n \in \texttt{parseTitle}(1, e) \Leftrightarrow$
$\quad (\exists e' : \texttt{HTMLTextElem} \bullet \texttt{content}(e') \neq \text{“}\text{“} \wedge \texttt{content}(e') = n) \vee$
$\quad (\exists e' : \texttt{HTMLElem} \bullet e' \in \texttt{content}(e) \wedge n \in \texttt{parseTitle}(1, e')) \wedge$
$i > 1 \Rightarrow \texttt{parseTitle}(i, e) = \{\}$

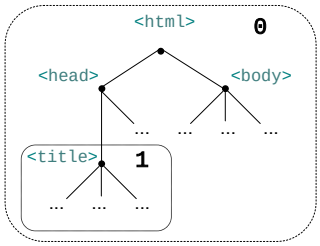


Figure 11.2: Specification for *BWeb* parser

`<title>` element and, hence, do not consider *concatenation* of text elements. Finally, we assume that element names contain lower case letters only. This is no limitation. Also, our JAVA-implementation includes a parser that assures this property when generating the formal data structures.

Now, we are ready to introduce the full specification of `parseBWeb`. It is listed in Fig. 11.2. We specify a tree parser that parses directory structures in three states and calls a parser for html document titles (`parseTitle`) that works in two states. The figures in the right-hand part relate parsing states of the respective parser to those areas of a well-formed structure (w.r.t. *BWeb* or html format) that they parse in the respective sate.

In state zero `parseBWeb` checks format requirement (1) from above (cf. Defn. 11.1.1); the directory d must exactly contain two directories “html” and “resources” as well as a file “index.html”. If `parseBWeb` succeeds in state zero it proceeds by parsing “html” and “resources” recursively in states one and two, respectively. There, requirements (3) and (4) are checked. Satisfaction of requirement (2) is verified by `parseTitle`. It remains in state zero until a `<title>` element is found. If no `<title>` element exists, `parseTitle` returns the empty set. Otherwise it returns the content of all non-empty textual content of `HTMLTextElem` elements that are sub-elements of `<title>`.

As an advantage, the specification in Fig. 11.2 is directly operational. A working

Table 11.2: *New functions introduced for* EntryPoint

Function / predicate			Implementing data type
validAWeb	:	`Website`	`Website`
validBWeb	:	`Website`	`Website`
containsDoc	:	`Dir` × `Doc`	`Dir`
containsElem	:	`HTMLElem` × `HTMLContElem`	`HTMLElem`
`parseBWeb`	:	`Nat` × `Dir` × `String` → `Bool`	`Website`
`parseTitle`	:	`Nat` × `HTMLElem` → `Set`[`String`]	`Website`

JAVA implementation could be deduced easily. Yet we remain to prove correctness of `parseWeb` and *validBWeb*. This is done in the proof of the next lemma. Also the lemma states that websites conforming to *BWeb* conform to *AWeb* as well.

Lemma 11.1.1 (*validBWeb* is sound) Given a website w : `Website`. Then we have:

(1) $validBWeb(w) \Rightarrow validAWeb(w)$.
(2) $validBWeb(w) \Leftrightarrow w$ conforms to *BWeb*.

□

The proof can be found in App. C.6, page 227.

This concludes the formal specification of EntryPoint. We have seen that our ADT-based approach suits well to handle tree structures, which frequently occur in our domain. As an example the tree parser `parseWeb` could be specified recursively in a straightforward way. Also, we were able to prove that it has the desired behavior. Before we briefly explain how EntryPoint can be registered in our prototype system, we collect those functions and predicates in Tab. 11.2 that have been introduced in this section. There, `size` and the data type `Nat` are not shown; they are standard. The functions collected in Fig. 11.2 indicate that iteration cycles between the analysis and design phase can in general not be avoided (particularly in large projects); we had to include new datatypes.

In Fig. 11.3 we show code snippets that specify and register EntryPoint and *AWeb* in our system. On the left-hand side the specification of EntryPoint starts by introducing a new concept symbol of arity three (lines `02`, `03`), which is non-functional (`false` parameter). Again, class `ArchSignature` administrates all concept symbols; registration is done in line `04`. Starting at line `07`, the roles of EntryPoint are registered. We use role `website` as an example. It is registered at position zero in the interface of EntryPoint and is treated like a regular variable (class `VarSymbol`). The type constraint (cf. Fig. 11.1) is set in line `17`. There, classes `BinOpFormula` and `InfixPredFormula` implement formulas that include binary boolean operators (like $\wedge$, $\vee$, $\Rightarrow$) and binary predicates in infix notation (like $=$, $<$), respectively. Also, they are sub-classes of class `Formula`, which is the super-class of all types of formulas that our system supports.

The right-hand part of Fig. 11.3 lists parts of the implementation of *AWeb*. First, a non-functional context symbol (class `NFContext`) `aWeb` is introduced in line `02`. Having prepared the parameter list in line `05`, which consists of a variable *website* : `Website` only, we can register the implementing formula (cf. Fig. 11.1). In particular, *AWeb* conformity $validAWeb(website)$ is represented by an instance of class `PredFormula` (line `12`), which implements predicate application. The condition $sourcedir = \texttt{srcDir}(website)$ is

Specification of *EntryPoint*

```
01 // define non-functional concept symbol
02 ConceptSymbol epc =
03   new ConceptSymbol("EntryPoint", 3, false);
04 // register concept
05 asig.addConcept(epc);
06 // set roles
07 epc.setParam(0,
08   new VarSymbol(
09     "website", new TypeVarSymbol("alpha_w")));
...                ...
16 // set type constraint
17 epc.setTypeConstraint(
18   new BinOpFormula(
19     DigArchiveSystem.BOOL_OPERATOR_AND,
20     new InfixPredFormula(
21       DigArchiveSystem.BOOL_OPERATOR_SUBTYPE,
22       new TypeVarSymbol("alpha_w"),
23       asig.getType("Website")
24     ),
25     new BinOpFormula(
...                  ...
35     )));
```

Specification of AWeb

```
01 // define non-functional context
02 ContextSymbol aWeb = new NFContext("AWeb");
03 // parameters for calling "srcDir"
04 ArrayList pars = new ArrayList ();
05 pars.add(new VarSymbol("website",
06  asig.getType("Website")));
07 // register implementation for AWeb
08 aWeb.addConceptImpl(
09   epc,
10   new BinOpFormula(
11     DigArchiveSystem.BOOL_OPERATOR_AND,
12     new PredFormula("validAWeb", pars),
13     new BinOpFormula(
14       DigArchiveSystem.BOOL_OPERATOR_AND,
15       new InfixPredFormula(
16         DigArchiveSystem.BOOL_OPERATOR_EQUAL,
17         new VarSymbol("sourcedir",
18                       asig.getType("Dir")),
19         new FunTerm("srcDir", pars)
20       ),
21       new InfixPredFormula(
...                    ...
25       ))));
```

Figure 11.3: Specification and registration of EntryPoint and *AWeb*

implemented at line 15. There, class `FunTerm` represents terms including function application (like `srcDir`(*website*)). When evaluating these terms, function calls are delegated to the respective JAVA methods that have been registered in advance.

11.2 The Concept AContent

The specification of AContent is shown in Fig. 11.4. Recall that AContent is *functional* and models content abstraction for non-html files (context *DocC*), html files (*HtmlDocC*), and directories (*DirC*). The upper left-hand part of Fig. 11.4 shows matches for AContent in our example website. Later on, we will use AContent to specify content preservation for these types of **container**s. In general, the interface (upper right-hand part of Fig. 11.4) permits arbitrary return types (α_t<`Top`).

Recall that content of html-files is to be preserved "as far as possible" only. In particular, html links will change during the transformation process. Therefore, we introduce an explicit *abstraction* function `abstrContElem` : `HTMLContElem` → `HTMLContElem`. The induced preservation scheme is shown in the bottom right-hand part of Fig. 11.4. When a source element of type `HTMLObjElem` (i.e., an `<a>` element in our example) is transformed to a new `HTMLObjElem`, its content is to be preserved *under abstraction* `abstrContElem`; the results of applying `abstrContElem` to the source and target element must be equal. In Fig. 11.4 we already indicate that `abstrContElem` sets the value of the `href` attribute to "" when applied to an `<a>` element. All other parts remain unchanged. The specification follows shortly.

Analogously, directory content is abstracted (lower left-hand part of Fig. 11.4); the implementation of *DirC* uses a function `abstrContent` : `HTMLDoc` → `AbstrDir`, which returns a tree-like structure (`AbstrDir`) that contains no files anymore. Recall that non-html and html content is strictly separated in *BWeb*. Directory structures in the source and target model will, thus, not be fully equal. The pure *directory* structure (containing

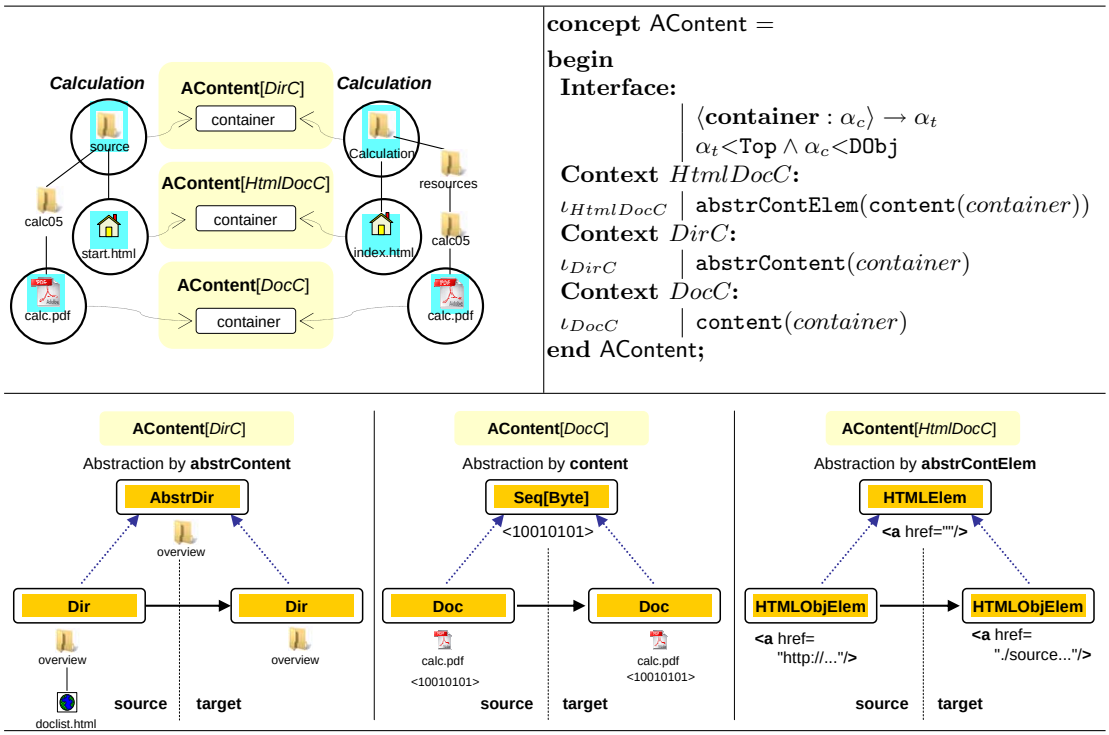


Figure 11.4: Specification of concept AContent

no files), however, is to be preserved. Both `AbstrDir` and `abstrContent` will be specified at the end of this section.

As we require preserving the *bit-wise* content of *non-html* files, *DocC* is implemented by `content`(*container*); *container* is of type `Doc`, and abstraction is done by `content`. The scheme is shown in the lower middle part of Fig. 11.4. File content is indicated by bit sequences. Abstraction is to extract this bit sequence here; the return type is set to `Seq`[`Byte`].

Abstraction as shown in Fig. 11.4 is a general design pattern in our approach: Define a datatype that abstracts from certain parts of the original datatype and specify related abstraction functions for the source and target context. The abstracted content then can be specified to be preserved using functional preservation constraints.

Specifying Abstraction of Html Content

Html content may change as we require websites in *BWeb* to contain *relative* links only. Other changes than that, however, are not accepted. The specification of `abstrContElem` in Fig. 11.5 defines a suitable class of abstraction functions and implements the preservation scheme in the lower right-hand part of Fig. 11.4. First, all basic content (text, CDATA, and comment elements) is preserved by `abstrContElem`. If *e* has complex content (i.e., *e* is an instance of `HTMLElem` or `HTMLObjElem`), abstraction is delegated to `abstrElem`. There, we treat abstraction of `<a>` elements separately. If *e* is an `<a>` element, attributes other than `href` have to be equally included in the abstracted element (cf. `abstrAttr`). The values of `href` attributes, however, are set to ““.

Specification for `abstrContElem` : `HTMLContElem` → `HTMLContElem`**:**

$\forall e$: `HTMLContElem` • $\neg$`inst`$_{\texttt{HTMLElem}}(e) \Rightarrow$ `abstrContElem`$(e) = e \wedge$
$\forall e$: `HTMLElem` • `abstrContElem`$(e) =$ `abstrElem`(e)

Specification for `abstrElem` : `HTMLElem` → `HTMLElem`**:**

$\forall e$: `HTMLElem` • $\neg$`inst`$_{\texttt{HTMLObjElem}}($`abstrElem`$(e))\wedge$
`name`(`abstrElem`(e)) = `name`$(e)\wedge$
(`name`$(e) =$ "`a`" $\Rightarrow \forall a$: `HTMLAttr` • `abstrAttr`$(a) \in$ `attrs`(`abstrElem`$(e)) \Leftrightarrow a \in$ `attrs`$(e))\wedge$
(`name`$(e) \neq$ "`a`" $\Rightarrow \forall a$: `HTMLAttr` • $a \in$ `attrs`(`abstrElem`$(e)) \Leftrightarrow a \in$ `attrs`$(e))\wedge$
$\forall i$: `Nat` • `get`(`content`(`abstrElem`$(e)), i$) = `abstrElem`(`get`(`content`$(e), i$))

Specification for `abstrAttr` : `HTMLAttr` → `HTMLAttr`**:**

$\forall a$: `HTMLAttr` • ((`name`$(a) \neq$ "`href`" $\Rightarrow$ `abstrAttr`$(a) = a) \wedge$
(`name`$(a) =$ "`href`" $\Rightarrow$ `abstrAttr`$(a) =$ `HTMLAttr`("`href`", ""))

Figure 11.5: Abstraction functions for html content

Apart from references in `<a>` elements, `abstrContent` : `HTMLDoc` → `HTMLElem` also abstracts from object identity; `abstrElem` does not contain any `HTMLObjElem`s. Otherwise, the abstracted content of a source and target document could not be equal; object IDs before and after a transformation are different.

Notice that preservation of html content w.r.t. `abstrContElem` does *not* assure that link anchors of the source document occur at the *same position* in the target document. This *structural* property will be modeled by Contains and Neighbor later on.

Recall that we want to preserve a website's title (requirement (5) in Fig. 8.1). When specifying concept EntryPoint in Sect. 11.1 we have already introduced the function `parseTitle`. When applied to an `HTMLElem` e it returns a set of strings corresponding to the content of all text elements that occur as sub-elements of a `<title>` element in e. Since `abstrContElem` changes no text elements and no `<title>` elements, it preserves the result of `parseTitle`; preserving the content of an html file w.r.t. `abstrContElem` includes preserving the title of that file. This will be important when preserving website titles later on (preservation requirement (5)).

Specifying Abstraction of Directory Structures

According to preservation requirement (9), directory and file structures have to be preserved. In particular, the source directory structure is to be duplicated; one copy is appended to the "html" directory and one is appended to the "resources" directory of the

Specification of `AbstrDir`**:**

```
type AbstrDir<{Top} =
begin
  constr [ AbstrDir : String × Set[AbstrDir] → @
  ops    [ name     : @ → String
           subDirs  : @ → AbstrDir
  axioms [ ∀n : String, sd : Set[AbstrDir]
             name(AbstrDir(n, sd) = n)∧
             subDirs(AbstrDir(n, sd) = sd
end AbstrDir;
```

Specification of `abstrContent`**:**

$\forall d$: `Dir` •
`name`(`abstrContent`(d)) = `name`$(d)\wedge$
$\forall d'$: `Dir` •
`abstrContent`$(d') \in$
`subDirs`(`abstrContent`$(d)) \Leftrightarrow$
$d' \in$ `subDirs`(d)

Figure 11.6: `AbstrDir` and specification of `abstrContent`

Table 11.3: *New functions introduced for* AContent

Function / predicate / type			Implementing data type
`abstrContElem`	:	`HTMLContElem` → `HTMLContElem`	`HTMLContElem`
`abstrElem`	:	`HTMLElem` → `HTMLElem`	`HTMLElem`
`abstrAttr`	:	`HTMLAttr` → `HTMLAttr`	`HTMLAttr`
		`AbstrDir`	
`abstrContent`	:	`Dir` → `AbstrDir`	`Dir`

target website. Since html- and non-html content is separated, our abstraction function considers directories only; preservation of *file* containment will be covered by concept Contains later on.

Abstracted directories will have type `AbstrDir`, which is specified in the left-hand part of Fig. 11.6. Type `AbstrDir` abstracts from object IDs and file-related content of directories; it merely stores directory names and sub-directories of type `AbstrDir`. The specification of `abstrContent` in the right-hand part of Fig. 11.6 is straightforward in this respect; whenever $\texttt{abstrContent}(d) = \texttt{abstrContent}(d')$, d and d' have equal names and contain equally named directories. It is easy to see that preserving `abstrContent` for a given directory implies preserving the name as well. In formal terms, $\mathsf{AContent}(d)[\mathit{DirC}] \Rightarrow \mathsf{Name}(d)[\mathit{DirN}]$ holds for all d; we need not check name preservation explicitly later on.

Again, we conclude the section by listing those data types and functions in Tab. 11.3 that have newly been introduced.

11.3 The Concept Contains

The specification of Contains is shown in Fig. 11.7. Contexts *CDir* and *CDoc* implement *direct* directory / file containment. *CDirRec* covers *recursive* directory containment. It calls *containsDir*, which works similar to *containsDoc*.

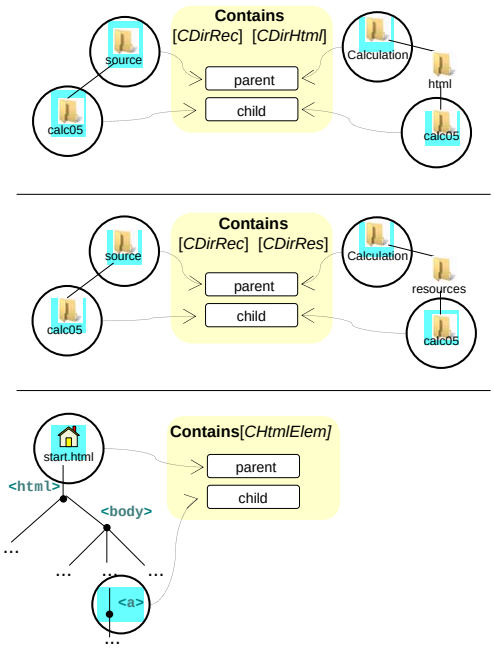
Contexts *CDirHtml* and *CDirRes* are particularly relevant for websites conforming to *BWeb*. When specifying preservation requirements later on, these contexts will be used as shown by the upper two visualizations in the left-hand part of Fig. 11.7. Our migration algorithm is required to generate two new versions of each directory of the source website — one copy for "html" and one copy for "resources". This yields two constraints:

(1) Whenever a directory is contained in the source directory of the source website (*CDirRec*), in must be contained in the "html" directory of the target website (*CDirHtml*, cf. upper visualization in Fig. 11.7).

(2) Whenever a directory is contained in the source directory of the source website (*CDirRec*), in must be contained in the "resources" directory of the target website (*CDirRes*, cf. middle visualization).

In the implementation of *CDirRec* we use a function `getSubDirByName`. When applied to a directory d, it returns the direct sub-directory of d, that has the respective name. If such a directory does not exists, `getSubDirByName` is undefined.

The contexts *CHtmlElem* and *CElemElem* deal with structure of html content. The former implements recursive element containment in an html document as is shown by



concept Contains =

begin

Interface:

	$\langle$**parent** : α_p, **child** : $\alpha_c\rangle$
	$(\alpha_p$<Dir $\wedge$ $(\alpha_c$<Dir$)$ $\vee$ α_c<Doc$)\wedge$
	$((\alpha_p$<HTMLDoc $\vee$ α_p<HTMLObjElem$)\wedge$
	α_c<HTMLObjElem$)$

Context $CDir$**:**

ι_{CDir}	$child \in$ subDirs$(parent)$

Context $CDoc$**:**

ι_{CDoc}	$child \in$ subDocs$(parent)$

Context $CDirRec$**:**

$\iota_{CDirRec}$	$containsDir(parent, child)$

Context $CDirHtml$**:**

$\iota_{CDirHtml}$	$containsDir($ getSubDirByName$($ $parent$, "html"$)$, $child)$

Context $CDirRes$**:**

$\iota_{CDirRes}$	$containsDir($ getSubDirByName$($ $parent$, "resources"$)$, $child)$

Context $CElemElem$**:**

$\iota_{CElemElem}$	$containsElem(parent, child)$

Context $CHtmlElem$**:**

$\iota_{CHtmlElem}$	$containsElem(parent, child)$

end Contains;

Figure 11.7: Specification of concept Contains

the lower visualization in Fig. 11.7; the latter covers recursive element containment in HTMLElems. Both use a predicate *containsElem* for this purpose.

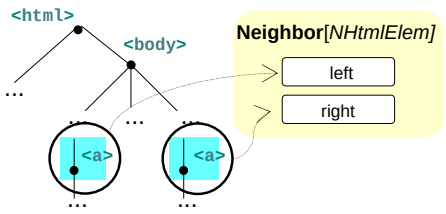
In Tab. 11.4 we show the signatures for the newly introduced functions of Fig. 11.7. All specifications are straightforward realizations of the functionalities described above and are omitted for brevity. Yet notice that getSubDirByName only works because all names of all sub-directories are mutually distinct (cf. Sect. 9.1).

11.4 The Concept Neighbor

The specification of Neighbor is shown if Fig. 11.8. In our example it occurs in the pattern visualized in the left-hand part; we trace <a> elements only. The implementation of *NHtmlElem* uses the predicate $isNeighbor$ the specification of which is shown in the bottom part of Fig. 11.8. It returns true iff e' is located somewhere below e in the html tree w.r.t. some parent element pe. This seemingly complicated implementation is necessary as we trace html structures only *partly*.

Table 11.4: *New functions introduced for* Contains

Function / predicate / type			Implementing data type
getSubDirByName	:	Dir $\times$ String $\rightarrow$ Dir	Dir
containsDir	:	Dir $\times$ Dir	Dir
containsElem	:	HTMLDoc $\times$ HTMLElem	HTMLDoc
containsElem	:	HTMLElem $\times$ HTMLContElem	HTMLContElem



concept Neighbor =
begin
Interface:

$$\begin{array}{l|l} & \langle \textbf{left} : \alpha_l, \textbf{right} : \alpha_r \rangle \\ & \alpha_l < \texttt{HTMLObjElem} \wedge \\ & \alpha_r < \texttt{HTMLObjElem} \end{array}$$

Context $NHtmlElem$**:**

$$\begin{array}{l|l} \iota_{NHtmlElem} & isNeighbor(left, right) \vee \\ & isNeighbor(right, left) \end{array}$$

end Neighbor;

Specification for $isNeighbor : \texttt{HTMLContElem} \times \texttt{HTMLContElem}$**:**

$$\begin{array}{l} \forall e, e' : \texttt{HTMLContElem} \bullet isNeighbor(e, e') \Leftrightarrow \\ \quad \exists i, j : \texttt{Nat}, pe : \texttt{HTMLElem} \bullet \\ \quad\quad i \neq j \wedge e = \texttt{get}(\texttt{content}(pe), i) \wedge \\ \quad\quad (e' = \texttt{get}(\texttt{content}(pe), j) \vee containsElem(\texttt{get}(\texttt{content}(pe), j), e')) \end{array}$$

Figure 11.8: Specification of concept Neighbor

Notice that the relative positioning of link anchors within a given html document is uniquely determined by Neighbor and Contains; preservation of these two concepts, thus, means preserving the link positioning within html documents.

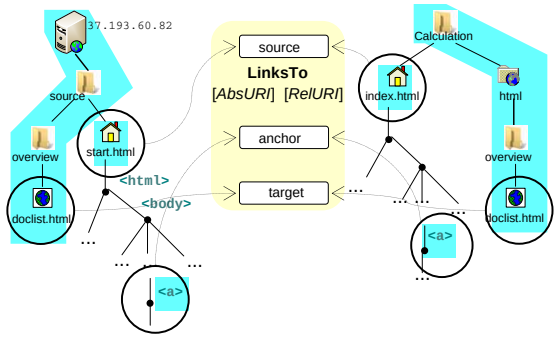
11.5 The Concept LinksTo

A general approach to incorporating graph-based querying has been introduced in Chap. 7 already. Html linking is an instance thereof. In Chap. 7 we have particularly introduced a grammar for well-formed URLs, the graph specification scheme for querying directory structures, and an example link automaton for absolute URLs. Therefore, we do not recapitulate the single parts here, but directly switch to the specification of LinksTo in Fig. 11.9. The contexts *AbsURI* and *RelURI* are visualized in the left-hand part. The shaded areas indicate that a server is included whenever *AbsURI* is matched. In contrast, relative URIs (*RelURI*) navigate relative to the link source.

For the sake of simplicity, we merely cover links in `<a>` elements and relative links containing no leading / (right-hand part of Fig. 11.9). Thus, both ι_{AbsURI} and ι_{RelURI} require the link anchor to have name "`a`". Also, it must be in the content of the link **source**. If this is true, both contexts use different acceptance conditions of the underlying automata. Concerning *AbsURI*, the predicate $acceptsIn(ha, ref, trg)$ is true iff the automaton ha accepts the word ref in a final state with semantics trg when starting at an arbitrary initial state. There, `AbsURIAutomaton(existDObj)` constructs a dominated product automaton that

- recognizes valid absolute URIs as has been explained in Chap. 7 and
- contains existing objects as states only.

In particular, we have used the graph specification in Fig. 7.4 on page 113. As automata are standard, we do not explicitly introduce the specification for `DomProductAutomaton`, which handles dominated product automata in our system. Notice, however, that both,



Specification for `getAttrByName`

$\forall e : \texttt{HTMLElem}, n : \texttt{String}, a : \texttt{HTMLAttr} \bullet$
$\texttt{getAttrByName}(e, n) = a \Leftrightarrow$
$(a \in \texttt{attrs}(e) \wedge \texttt{name}(a) = n)$

concept LinksTo =

begin

Interface:

$\langle \textbf{source} : \alpha_s, \textbf{anchor} : \alpha_a, \textbf{target} : \alpha_t \rangle$
$\alpha_s < \texttt{HTMLDoc} \wedge \alpha_a < \texttt{HTMLObjElem} \wedge \alpha_t < \texttt{Doc}$

Context $AbsURI$**:**

ι_{AbsURI} | $\texttt{name}(anchor) = \text{“}\texttt{a}\text{“} \wedge$
$containsElem(source, anchor) \wedge$
$acceptsIn($
$\texttt{AbsURIAutomaton}(\texttt{existObj}),$
$\texttt{value}($
$\texttt{getAttrByName}(anchor, \text{“}\texttt{href}\text{“})),$
$target)$

Context $RelURI$**:**

ι_{RelURI} | $\texttt{name}(anchor) = \text{“}\texttt{a}\text{“} \wedge$
$containsElem(source, anchor) \wedge$
$\exists d : \texttt{Dir} \bullet$
$source \in \texttt{subDocs}(d) \wedge$
$accepts($
$\texttt{RelURIAutomaton}(\texttt{existObj}),$
$d,$
$\texttt{value}($
$\texttt{getAttrByName}(anchor, \text{“}\texttt{href}\text{“})),$
$target)$

end LinksTo;

Figure 11.9: Specification of concept LinksTo

`AbsURIAutomaton` and `RelURIAutomaton` are subtypes of `DomProductAutomaton`; we implement graph specification schemes by refining `DomProductAutomaton`. As automata are regular datatypes, concept specifications remain short and readable (cf. Fig. 11.9); at the same time, we do not loose the constructive semantics of automata.

Since servers are initial states in `AbsURIAutomaton`, *acceptsIn* does not need an explicit source node for being evaluated. The function `getAttrByName` selects the `href` attribute and is specified in the lower left-hand part of Fig. 11.9. It is undefined if no suitable attribute exists. This specification works since we have required attribute names to be mutually distinct.

Relative links are evaluated w.r.t. a concrete source object, which is implemented by *accepts*. There, $accepts(ha, src, ref, trg)$ holds iff there is a run in ha starting at an initial state with semantics src and ending at a final state with semantics trg while recognizing ref. Hence, LinksTo$(src, anc, trg)[RelURI]$ holds if the value of the `href` attribute of anc leads from the parent directory of src to trg.

Notice that ι_{AbsURI} and ι_{RelURI} instantiate `AbsURIAutomaton` and `RelURIAutomaton`, respectively, using `existDObj`. This is inefficient if done each time LinksTo is evaluated.

Table 11.5: *New functions introduced for* LinksTo

Function / predicate / type			Implementing data type
`AusURIAutomaton`, `RelURIAutomaton` < `DomProductAutomaton`			
`AusURIAutomaton`	:	`Set[DObj]` → `AbsURIAutomaton`	`AbsURIAutomaton`
`RelURIAutomaton`	:	`Set[DObj]` → `RelURIAutomaton`	`RelURIAutomaton`
acceptsIn	:	`DomProductAutomaton` × `String` × `DObj`	`DomProductAutomaton`
accepts	:	`DomProductAutomaton` × `DObj` × `String` × `DObj`	`DomProductAutomaton`
`getAttrByName`	:	`HTMLElem` × `String` → `HTMLAttr`	`HTMLElem`

Therefore, we administrate automata *iteratively* (w.r.t. system evolution) on the implementation level. This is a built-in feature of our prototype implementation. Future work might include formal integration of this feature into our framework. We conclude by listing those datatypes, functions, and predicates in Tab. 11.5 that have been introduced here. On the whole, we now have all concepts available that are needed to formulate our preservation requirements formally.

Chapter 12

Specifying and Checking Preservation Requirements

Implementing concepts and contexts requires the most effort when applying our approach; technical implementation details can be challenging. In contrast, preservation requirements hide implementation details, which keeps formulations short and readable. In the following we provide formal preservation constraints for the running example and evaluate runtime costs for checking our migration algorithm against these constraints.

12.1 Formal Preservation Requirements

We separate four blocks: (1) Format preservation, (2) preservation of file and directory structure, (3) preservation of file content, and (4) preservation of link consistency.

Website Format

Website formats have been modeled by EntryPoint; the related preservation constraint is given as follows and fully covers preservation requirement (1) of Fig. 8.1:

(1) $\forall w : \texttt{Website} \bullet$
$\forall d : \texttt{Dir} \bullet$
$\forall h : \texttt{HTMLDoc} \bullet$
$\mathsf{EntryPoint}(w, d, h)[AWeb] \Rightarrow \textcircled{\triangledown}\texttt{pres}_{nf}(\mathsf{EntryPoint}(w, d, h)[AWeb, BWeb])$

Recall that $validBWeb(w) \Rightarrow validAWeb(w)$ for all websites w (Lemma 11.1.1, page 140). Hence, the antecedent in the implication above covers both, websites conforming to *AWeb* and such conforming to *BWeb*. If the source website is in format *BWeb* already, no transformation is required. If it is in *AWeb* and not in *BWeb*, a transformation is implicitly enforced. There, we require only "positive" preservation. If $\mathsf{EntryPoint}(w, d, h)[AWeb]$ does not hold, we do not require preservation of that status; we are interested in transformation results of valid websites only.

Since preservation formulas are treated as regular formulas in our prototype system, setting them up and registering them is quite similar to concept implementations (cf. Fig. 11.3). Therefore, we omit an explicit listing.

Directory Structures

Directory structures are covered by AContent and Contains; we provide three preservation formulas that — when adhered to — assure that the target website (in *BWeb* format) contains the source directory structures as sub-structures of "html" and "resources".

(2) $\forall d : \texttt{Dir} \bullet$
$\neg(\exists w : \texttt{Website} \bullet \exists h : \texttt{HTMLDoc} \bullet \mathsf{EntryPoint}(w, d, h)[AWeb]) \Rightarrow$
$\textcircled{\forall}\texttt{pres}_f(\mathsf{AContent}(d)[DirC, DirC])$

(3) $\forall parent : \texttt{Dir} \bullet$
$(\exists w : \texttt{Website} \bullet \exists h : \texttt{HTMLDoc} \bullet$
$\mathsf{EntryPoint}(w, parent, h)[AWeb] \wedge \neg\mathsf{EntryPoint}(w, parent, h)[BWeb]) \Rightarrow$
$\forall child : \texttt{Dir} \bullet$
$\textcircled{\exists}\texttt{pres}_{nf}(\mathsf{Contains}(parent, child)[CDirRec, CDirHtml]) \wedge$
$\textcircled{\exists}\texttt{pres}_{nf}(\mathsf{Contains}(parent, child)[CDirRec, CDirRes]) \wedge$
$\textcircled{\forall}\texttt{pres}_{nf}(\mathsf{Contains}(parent, child)[CDirRec, CDirHtml]) \vee$
$\texttt{pres}_{nf}(\mathsf{Contains}(parent, child)[CDirRec, CDirRes])$

(4) $\forall d : \texttt{Dir} \bullet$
$(\exists w : \texttt{Website} \bullet \exists h : \texttt{HTMLDoc} \bullet \mathsf{EntryPoint}(w, d, h)[AWeb]) \Rightarrow$
$\forall d' : \texttt{Doc} \bullet$
$\mathsf{Contains}[CDoc](d, d') \Rightarrow (\textcircled{\exists} d' \mapsto \texttt{Doc} \Rightarrow \texttt{pres}_{nf}(\mathsf{Contains}(d, d')[CDoc, CDocRec])) \wedge$
$\neg(\exists w : \texttt{Website} \bullet \exists h : \texttt{HTMLDoc} \bullet \mathsf{EntryPoint}(w, d, h)[AWeb]) \Rightarrow$
$\forall d' : \texttt{Doc} \bullet \textcircled{\exists} d' \mapsto \texttt{Doc} \Rightarrow \texttt{pres}_{nf}(\mathsf{Contains}(d, d')[CDoc, CDoc])$

Constraint (2) requires preservation of sub-directory structures. It is applied to those directories only that are no source directory of any website; in the other case a change is enforced by introducing "html" and "resources". Again, we do not enforce transformations; constraint (2) always holds if no transformations take place.

The third constraint enforces duplication of the source structure if *parent* is the entry point of a website that is valid, but not yet in format *BWeb*; websites conforming to *BWeb* remain untouched. If *parent* is an appropriate directory, another directory *child* is selected. If it is a sub-directory of *parent*, we require two traces to exist. The first trace assures that a copy of *child* is made and appended to the "html" directory in the target structure. Similarly, the second trace requires a copy for "resources". The last two lines of constraint (3) assure that the transformation process generates no other copies than these two.

Preservation of direct file containment is required by constraint (4). There, some special cases must be considered. First, we distinguish source directories of websites (EntryPoint holds in this case) and other directories (EntryPoint does not hold). When transforming a directory in the latter case, there must be a trace such that direct file containment is preserved. When transforming a directory in the former case, direct containment cannot be preserved; either d' is located in the transformed source directory (then $d' = h$) or it is located in "html" or "resources", respectively. Notice that d' cannot be located in other than these three directories as (1) all directories of the source website have an ID (2) their transformations are traced, and (3)

$$(\textcircled{\exists} d' \mapsto \texttt{Dir} \Rightarrow \texttt{pres}_f(\mathsf{Contains}(d, d')[CDoc, CDoc]))$$

holds for all d that are different to the website's source directory. In particular, a directory cannot contain a file in the target structure if its transformation source did not contain the transformation source of that file in the source structure; "html", "resources" and the new website source directory are the only directories not affected by this constraint.

Also, notice that the prerequisite $\mathsf{Contains}(d, d')[CDoc]$ is important in the first part of constraint (4) (d is the source directory of a website). Otherwise we would prohibit the transformation result of d' to be recursively contained in the source directory of the result website. This does not hold in our example.

On the whole, constraints (2) to (4) cover preservation requirements (3) and (9) of Fig. 8.1. The detailed explanations indicate that care has to be taken as soon as case distinctions occur. Indeed our initial formulation of constraint (4) did not meet the intended preservation requirement.

File Contents

Here we distinguish non-html files and html files; the former are subject to bitwise content preservation (requirement (8) in Fig. 8.1). In contrast, html content is to be preserved under abstraction `abstrContent`. Also, positioning of link anchors must be maintained.

(5) $\forall d : \texttt{Doc} \bullet \neg \texttt{inst}_{\texttt{HTMLDoc}}(d) \Rightarrow$
$\quad \textcircled{\forall}\texttt{pres}_f(\mathsf{Name}(d)[DocN, DocN]) \wedge \texttt{pres}_f(\mathsf{AContent}(d)[DocC, DocC])$

(6) $\forall parent : \texttt{HTMLDoc} \bullet$
$\quad \texttt{pres}_f(\mathsf{AContent}(parent)[HtmlDocC, HtmlDocC]) \wedge$
$\quad \forall child : \texttt{HTMLObjElem} \bullet$
$\quad\quad \textcircled{\forall}\texttt{pres}_{nf}(\mathsf{Contains}(parent, child)[CHtmlElem, CHtmlElem])$

(7) $\forall e : \texttt{HTMLObjElem} \bullet$
$\quad \forall e' : \texttt{HTMLObjElem} \bullet$
$\quad\quad \textcircled{\forall}\texttt{pres}_{nf}(\mathsf{Contains}(e, e')[CElemElem, CElemElem]) \wedge$
$\quad\quad\quad \texttt{pres}_{nf}(\mathsf{Neighbor}(e, e')[NHtmlElem, NHtmlElem])$

Constraint (5) fully assures preservation requirement (8) as $\mathsf{AContent}(d)[DocC]$ returns the bit-wise content of d. Also, it assures requirement (2) partly. Constraint (6) assures preservation of html content w.r.t. `abstrContent` and html element containment. This partly realizes requirement (7). Recall that `abstrContent` preserves the output of `parseTitle`; constraint (6) subsumes preservation of website titles and, hence, covers requirement (5) as well. Again, we explicitly mention that

$$\texttt{pres}_{nf}(\mathsf{Contains}(parent, child)[CHtmlElem, CHtmlElem])$$

requires preservation of the *status* of containment; the result file must not contain html elements that were not in the source file.

It remains to constrain positioning of link anchors. We achieve this by preserving mutual containment and neighborship in constraint (7). There, it is important that *all* link anchors are digital objects; otherwise they are not covered by constraint (7). If both constraints (6) and (7) hold for given source and target html files, these files differ in their html links at maximum.

Link Consistency

Similar to format preservation, the necessary implementation work is the most challenging part; the requirement itself is easily stated and fully covers preservation requirement (6) of Fig. 8.1.

(8) $\forall src : \texttt{HTMLDoc} \bullet$
$\forall anc : \texttt{HTMLObjElem} \bullet$
$\forall trg : \texttt{Doc} \bullet$
$\mathsf{LinksTo}(src, anc, trg)[\_] \Rightarrow \textcircled{\forall}\texttt{pres}_{nf}(\mathsf{LinksTo}(src, anc, trg)[\_, RelURI])$

The wildcard in the antecedent of the implication and in the non-functional preservation constraint itself keeps the specification short. Recall that it causes *all* contexts of LinksTo to be checked. In our example, the constraint indeed has to be applied whenever an arbitrary link is found. The output of a link transformation, however, is constrained to be a relative URI.

12.2 Evaluating Runtime Costs

System performance is important when aiming at practical acceptance. In general, however, high performance w.r.t. computation time and high trustworthiness of underlying methods are mutually limiting goals. Related to our example, there are a lot of link consistency checkers available that work quite efficiently. Yet their functional correctness and reliability are difficult to estimate. Also — to the best of our knowledge — there are no tools available that are capable of tracing object histories and checking formal preservation requirements in the fashion described in this thesis. When standard link consistency checkers report link consistency this does not mean preservation in our sense; it simply reports that all found links point to valid locations.

The measurements in this section mainly reflect runtime costs of *formal object tracing* and *formal property checking*. We are well aware that cutting down the claim for formal correctness results in better runtime performance. However, we shall see that runtime costs strongly depend on the implementation and formulation of preservation requirements; relevant model sizes and properties can be handled when using advantageous constraint formulations.

Preparations and Test Environment

In order to have configurable test data, we have implemented a website generator that is generic in

(1) the number d of directories,
(2) the number h of html files,
(3) the number nh of non-html files,
(4) the number l of links *per html file*, and
(5) the number e of additional html elements per html file.

The underlying directory and file structure is generated randomly. Included links are valid *absolute* URIs w.r.t. RFC 2396; the transformation process indeed has to generate new *relative* links. Test results in the next section relate to different configurations.

All tests have been carried out on a Windows XP workstation with 1 GB RAM and an Intel P4 CPU with 3.2 GHz. Test websites have been deployed on an Apache Web Server version 2.2. All caching has been turned off in our system in order to achieve unadulterated results.

The underlying migration algorithm has been introduced in Chap. 10. As it is implemented in our functional language, it is operational immediately in our system. In general, however, the underlying migration algorithm has to be connected to our system. There, transformations are executed using registered transformation classes; each single transformation step must be communicated. Tracing has to be activated in advance. Upon completion of the transformation, constraint checking has to be activated; this immediately starts constraint checking and report generation. The resulting report contains a detailed summary of elapsed time (for each single constraint) and constraint violations; violations always refer to concrete object IDs.

Test Configurations and Results

On the whole we have set up three test scenarios. We deal with each of them separately in the following subsections.

Scenario One

In our example LinksTo is computationally the most complex concept; we exclude it for the time being and consider it separately in the subsequent scenarios.

Here, we compare computation times for checking preservation constraints (1) to (7) w.r.t. different website configurations. The results are shown in Fig. 12.1.

The $x-$ and $y-$ axis carry the used website configuration and the time in seconds, respectively. Website configurations are denoted as tuples (d, h, nh, l, e) as described in the last section. Hence, the website for the left-most configuration contains 50 directories, 50 html files, 100 non-html files, 50 links (one per html file), and an additional load of 100 html elements per html file. From left to right we have increased configurations by 50 directories, 50 html files, and 100 non-html files in each step; as the number of links and the number of additional html elements are defined *per file*, their amount increases accordingly. This constantly increases the overall number of objects in the system by 250. Website sizes range from $0,5$MB to $3,5$MB.

The top-most line in Fig. 12.1 shows the overall computation time. This time includes tracing, evaluation times for constraints (1) to (7), and system overhead (recall that our system assures some invariants). It does *not* include parsing (i.e., generating the formal website model) and serializing the transformation result. While constraint (5) (preservation of name and content of non-html files) requires the least computation time, constraints (6) (html file content AContent, Contains), (4) (direct file containment Contains), and (7) (structure of html content Contains, Neighbor) are the most time consuming ones. This is not surprising; they include non-functional preservation constraints

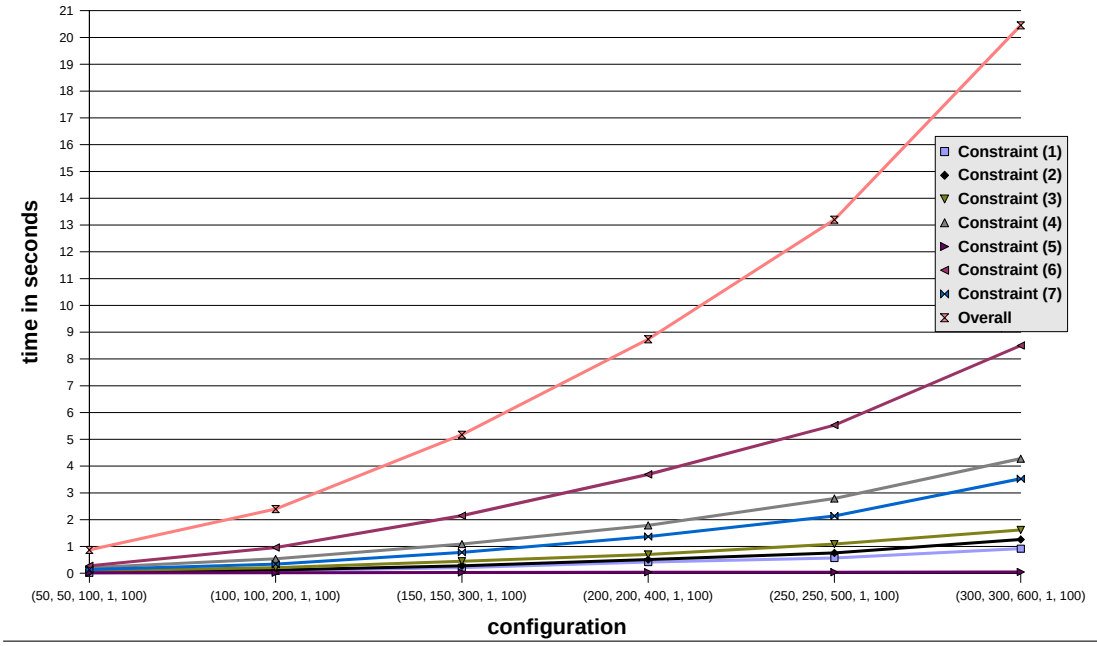


Figure 12.1: Runtimes for constraint checking, excluding LinksTo

the arguments of which are universally quantified. A considerable number of matches can be found in the respective models.

The overall behavior is linear w.r.t. the number of objects in the system. Indeed, we do not expect runtime performance to behave better than that; constraints (1) to (7) cover all parts of the source and target website. In Tab. 12.1 we show detailed results for the overall performance. There, we relate configurations #1 to #6 to the resulting number of objects in the system and the overall evaluation time for constraints (1) to (7) (cf. Fig. 12.1). In particular, we evaluate two quotients: The first one divides the number of objects for configuration i by the number of objects for configuration $i-1$. The other one relates evaluation times for configuration i and $i-1$. On average, the overall evaluation time grows with factor $1,32$ compared to the growth of number of objects in the system (cf. last column of the table). This factor contains two sources of overhead — system overhead and redundancy in the preservation constraints. Nevertheless, the results confirm the expected linear correlation between model sizes and overall computation times. Also, we conclude that constraints (1) to (7) contain acceptable redundancy as, otherwise, the factor 1.32 would be worse.

Table 12.1: *Overall performance excluding* LinksTo

Configuration	Objects in system	Time	Quotient 1	Quotient 2	Factor
#1 : (50, 50, 100, 1, 100)	250, 00	0, 87	–	–	–
#2 : (100, 100, 200, 1, 100)	500, 00	2, 4	2, 00	2, 76	1, 38
#3 : (150, 150, 300, 1, 100)	750, 00	5, 17	1, 50	2, 16	1, 44
#4 : (200, 200, 400, 1, 100)	1000, 00	8, 74	1, 33	1, 69	1, 27
#5 : (250, 250, 500, 1, 100)	1250, 00	13, 2	1, 25	1, 51	1, 21
#6 : (300, 300, 600, 1, 100)	1500, 00	20, 45	1, 20	1, 55	1, 29
		average:	1, 46	1, 93	1, 32

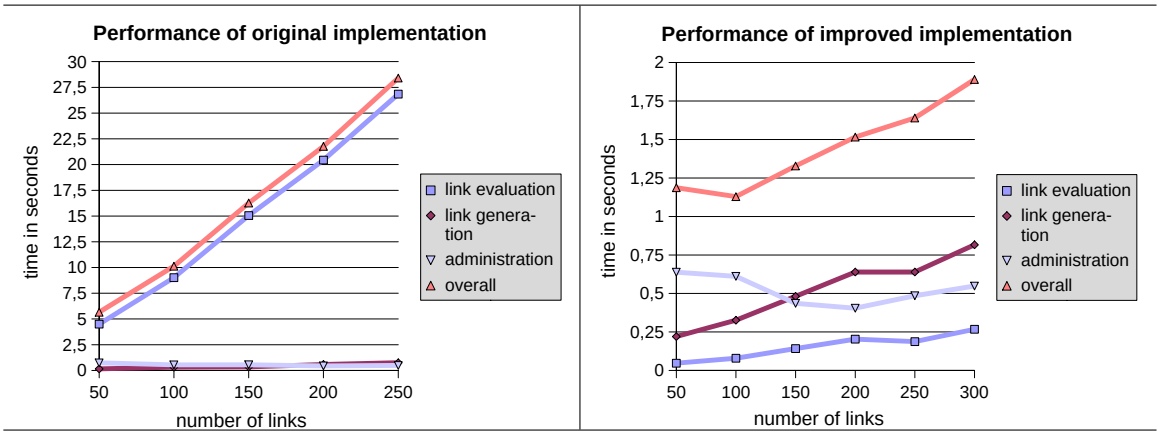


Figure 12.2: Performance for link processing w.r.t. original and improved implementation

Scenario Two

As explained in Chap. 7, link consistency represents a broad and relevant class of properties in our application domain. Among others, our methods can be used to generate document layout, document structures, and other graph-related queries like XPath. Therefore, we examine the computational behavior w.r.t. LinksTo in detail in the following two scenarios.

Compared to the other concepts, LinksTo potentially has the most matches in our example. A website of configuration (d, h, nh, l, e) has $h \cdot l \cdot (h + nh)$ potential matches for LinksTo; in configuration #1, e.g., $50 \cdot 150 = 7500$ checks are performed when computing the antecedent of constraint (8). This results in high runtime costs as is shown in the left-hand part of Fig. 12.2. There, we have used configuration #1 as a starting point and increased the number of links per html file by one in each step. This resulted in 50 links in the smallest configuration and 250 links in the largest one. Also, we have used formal link evaluation *and* link generation techniques as explained in Chap. 7.

The top-most line in Fig. 12.2 shows the overall computation time. It includes (from top to bottom) times for formal link evaluation, link generation, and administration of link automata. There, the administrative overhead for link automata is negligible; it is a constant factor since the model size does not change. Also, link generation shows the expected performance; generating 250 links in this (comparatively small) model took $0,75$ seconds.

In contrast, link evaluation shows poor performance, which results in unacceptable runtimes for configurations #2 to #5. While the functional behavior is linear w.r.t. the number of links in the system, link evaluation took $26,84$ seconds in the largest model. This is due to the high number of needless tries; in the largest model (250 links), our system performed more than 100000 link evaluations.

On the right-hand side of Fig. 12.2 we show the performance when using the following constraint, which is equivalent to constraint (8) but minimizes the number of link evaluations:

$$
\begin{array}{ll}
(8') & \forall src : \texttt{HTMLDoc} \bullet \\
& \quad \forall anc : \texttt{HTMLObjElem} \bullet anc \in \texttt{contentElems}(src) \Rightarrow \\
& \quad\quad \forall trg : \texttt{Doc} \bullet trg \in \texttt{linkTargets}(src, anc) \Rightarrow \\
& \quad\quad\quad \textcircled{\forall} \texttt{pres}_{nf}(\mathsf{LinksTo}(src, anc, trg)[\_, RelURI])
\end{array}
$$

There, two standard techniques are used, the benefit of which has been pointed out in related works already (e.g., [Sch04]). First, we use *set-valued functions* `contentElems` and `linkTargets` that compute those combinations of *src*, *anc*, and *trg* that definitely match LinksTo. In particular, `contentElems` returns all `<a>` elements (recall: we trace no other html elements than these) and `linkTargets`(src, anc) returns a set containing exactly those documents *anc* points to; in our example this set is empty or a singleton. Since we support ADTs and formal specification techniques, we do not loose the formal underpinning; we maintain a high degree of trustworthiness.

Second, quantification scopes are minimized in constraint (8'). Roughly, quantifiers are put inward as far as possible. Also, the quantors are "guarded", which avoids unnecessary computations. A more sophisticated technique, which subsumes minimizing quantification scopes, is known as *miniscoping* in the literature ([Wan60, de 86]).

The gain in performance is obvious in the right-hand part of Fig. 12.2. In particular, the overall linear behavior w.r.t. the number of links in the system is confirmed and link evaluation is less time consuming than link generation. Using constraint (8') we reduce the maximum number of link evaluations to $5 \cdot 50 = 250$ in the least configuration and $5 \cdot 250 = 1250$ in the largest of; we speak of a maximum number since wildcard evaluation stops as soon as a match is found. Also, `linkTargets` checks both, relative and absolute links. Hence, $3 \cdot 50 = 150$ and $3 \cdot 250 = 750$ link evaluations are necessary in the best case.

Notice, however, that the techniques above are applicable only because of our tolerant formulation of link consistency; we require *existing* links to be maintained but do not care about non-existing links. If we required preservation of the latter status as well, evaluating LinksTo for all possible combinations of link sources and link targets would be mandatory.

Apart from that, constraint (8') exposes implementation details of LinksTo whereas constraint (8) does not. Therefore, we do not enforce the implementation pattern in constraint (8') but mark automated miniscoping for future work. Also, we admit that performance improvements should be taken into account when planning time schedules for developing and implementing preservation requirements; several iteration cycles may be necessary in order to achieve acceptable runtimes.

Scenario Three

Here we check system performance when evaluating LinksTo in the configurations of scenario one — the underlying model sizes are practically more relevant than those used in scenario two. The results are shown in Fig. 12.3. The overall computation time ranges from $0,92$ to $9,47$ seconds. All parts show linear behavior w.r.t. the number of objects in the system. Concerning administrative overhead this cannot be expected in general; it includes adapting automata to model evolutions. Link graphs, however, are *sparse*, i.e., the number of edges is considerably smaller than $|V|^2$, where $|V|$ is the number of vertices (cf. Fig. 7.6, page 119). Adding a file, e.g., causes a *local* update of

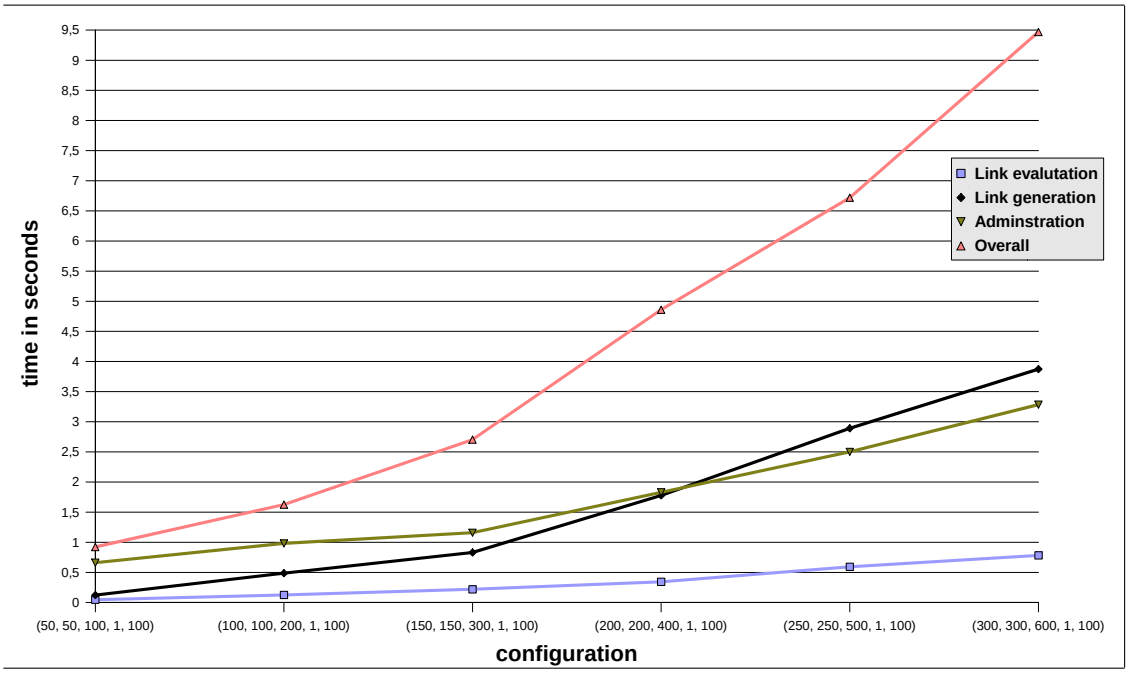


Figure 12.3: Runtimes for checking LinksTo

the related automaton only; one state and one transition has to be added. The resulting good performance cannot be expected in presence of highly connected structures. In our application domain, however, *tree* structures and sparse graph structures have various applications such that it seems to be sensitive to consider using our automata-based methods in these areas.

Tab. 12.2 shows detailed results for the overall performance w.r.t. LinksTo. The columns from left to right carry the following information:

Config.:	The configuration.
Objects:	The number of objects in the source state.
Link eval.:	Overall time for link evaluation.
Link gen.:	Overall time for link generation.
Quot. 1:	Ratio of numbers of objects in configurations $\#i$ and $\#(i-1)$.
Quot. 2:	Ratio of link *evaluation* times in configurations $\#i$ and $\#(i-1)$.
Quot. 3:	Ratio of link *generation* times in configurations $\#i$ and $\#(i-1)$.
Factor 1:	Growth rate of link *evaluation* times compared to growth rate of number of objects in the system.

Table 12.2: *Overall performance for* LinksTo

Config.	Objects	Link eval.	Link gen.	Quot. 1	Quot. 2	Quot. 3	Factor 1	Factor 2
#1	250	0,05	0,12	-	-	-	-	-
#2	500	0,13	0,49	2,00	2,72	3,96	1,36	1,98
#3	750	0,22	0,83	1,50	1,75	1,70	1,17	1,14
#4	1000	0,34	1,78	1,33	1,57	2,14	1,17	1,61
#5	1250	0,59	2,89	1,25	1,73	1,63	1,38	1,3
#6	1500	0,78	3,87	1,20	1,32	1,34	1,1	1,12
			average:	1,46	1,82	2,15	1,27	1,51

Factor 2: Growth rate of link *generation* times compared to growth rate of number of objects in the system.

On average, link evaluation grows with factor $1,27$ (before-last column) compared to the model size; indeed a factor close to 1 can be expected here. Link generation times grow with factor $1,51$ (last column) on average and w.r.t. the number of objects in the system. Again, considerable worse behavior up to $O(|V|^2)$ can be expected if the underlying graphs (automata) are highly connected.

All in all, the above linear behavior is promising. Keeping in mind that there is still potential for improvements, we consider the runtimes above to be acceptable. After all, runtimes include *generation and evaluation*. Also, checking LinksTo in the target model might be considered to be superfluous — we can prove that links are working and point to the desired location when our formal link generation techniques are used.

Chapter 13

Summary — Costs and Benefits

On the whole, this case study together with the one conducted in [Bor07] let us conclude the following:

- Our methods are both suitable and beneficial for our application domain. Our preservation language together with the ADT-based approach offer sufficient expressiveness for handling relevant classes of properties. Our automata-based approach could be used to handle link consistency in a fully automated way.
- It is important to select those properties that are worth being treated formally as concept specifications can require considerable effort.
- Our measurements show that moderately sized website models can be handled with acceptable performance (largest model: 300 directories, 300 html files, 600 non-html files, 300 html links, 100 html elements per html file).
- We assess the runtime performance to be sufficient to support the *development* of transformation algorithms as tests can be conducted with models that are small, but highly aligned.
- We advocate an interactive process when specifying preservation requirements and implementing transformations. In our example, we found both, overspecifications in the preservation requirements and functional errors in the transformations. The precise reports generated by our system helped here.

The following parts provide some more details.

Datatype Specification

Basic datatype specifications (i.e., datatypes including attributes only) required minor effort; they are related to XML data in our example and are, thus, largely standard. Also, they are re-usable. In our application domain XML data is used frequently, be it in digital objects themselves or as meta data. XML is tree-structured, which results in comfortable and short data models; (recursive) ADTs are well-suited for this domain. Related techniques (like tree parsers) can easily be specified and implemented (cf. `parseTitle`).

However, XML formats are quite "verbose". This has brought up the necessity to reduce the number of traced nodes in our example. Clearly, this is a general issue; needless tracing slows down system performance considerably. We have shown an example

and have suggested a general design pattern for tackling this issue. Since we had to trace `<a>` elements only, we have introduced an explicit object type `HTMLObjElem`, which is a subtype of the *basic* type `HTMLElem`; html content has been modeled by basic types in the first instance; only `<a>` elements are covered by `HTMLObjElem`. Our support for multiple inheritance is important here; `HTMLObjElem` is a subtype of both, `HTMLElem` and `DObj HTMLObjElem`.

The datatypes in our case study model *static* content only. This keeps our models simple but neglects document dynamics; in our example, no html forms and related actions could be modeled. Extending our framework into this direction is surely a topic of future research.

Concept Specification

This part has been the most time-consuming one. We have propagated keeping concept specifications short by outsourcing property checking to functions and predicates; they are compiled in our system, which speeds up constraint checking considerably. While the resulting concept specifications are short, specifying corresponding predicates and functions and deriving a reliable implementation may be quite involved. We propose to keep the step from function specifications towards running implementations small. This supports confidence in the derived implementations but may require for formal refinement steps. Using formal ADTs is beneficial here; refinement processes can be supported by theorem provers and other related tools. Also, tree-like structures as they occur in our system can naturally be handled quite well by ADTs. Due to the formal underpinning we could, e.g., prove that our *BWeb* parser works correctly.

The concepts that have been identified in the case study have been quite general. Containment Contains and other structure, content AContent, format validity, and graph-based querying LinksTo, e.g., occur in many other contexts. This favors re-use and decreases development times when concept libraries grow larger. Particularly, our automata-based approach is promising. As automata are regular datatypes, concept specifications remain short and readable. At the same time, we do not loose the constructive semantics. The proper choice of "semantic" non-terminals of the underlying grammar is vital (cf. Chap. 7). In general, this requires expert-level knowledge of the approach and of the application domain. Concerning the URI specification [Int98], however, the mapping to corresponding parts of the formal datatypes for servers, directories, and documents could be done quickly. Setting up the graph specification has been straightforward then. Finally, the underlying automata are constructed automatically such that specifications are operational immediately.

Implementing the Transformation Process

When implementing transformations some knowledge about our system is necessary. It is particularly important to distinguish between object creation and object transformation. Concept interfaces are a good indicator. Whenever objects are subject to preservation (i.e., they match a concept), they have to be be transformed. Otherwise their histories are not traced. This shows that, in general, specifying preservation requirements and implementing transformations should be carried out interactively. Also, deleting objects

requires some system-related knowledge. Objects have to be deleted top-down due to our system invariants. Whenever an object belongs to the content of another object, it cannot be deleted. As transformations get complex, it may be useful to build up intermediate structures. There, it can be quite involved to determine when corresponding objects can be deleted. We faced this issue when implementing our example algorithm. While support in this respect is surely desirable in future extensions, we argue that the importance of assuring object immutability outweighs the just-described difficulties.

Apart from that, our transformation algorithm has benefited from the model construction facilities of our automata-based approach; links could be generated automatically. There, access to the dynamic parts of our system is vital; link automata had to be instantiated w.r.t. `existDObj`.

Constraint Formulation

The concise requirements formulations in Chap. 12 satisfy our demand for readability. To our experience, however, constraint formulation gets involved when case distinctions come into play. Different variants of directory and file containment resulted in different preservation requirements in our example; the resulting constraints (3), (4) were not easy-to-formulate. Yet these constraints have helped in developing the transformation; a functional error could be found in earlier versions of the algorithm. Clearly, formal constraint formulation is a task for experts.

Concerning runtime costs, non-functional preservation constraints tend to be more time-consuming than functional preservation constraints; link consistency and containment (in different variants) were the most expensive properties. There, performance improvements can be achieved by using advantageous implementations. Three potential techniques for gaining efficiency are:

(1) Use functions and predicates to determine properties.
(2) Miniscope constraints.
(3) Combine constraints.

The potential performance gain when using techniques (1) and (2) has been shown using link consistency. Yet, miniscoping exposes implementation details, which may be declined by users. Combining constraints reduces evaluation overhead as variables and terms can be shared. It, however leads to less detailed reports and less readable specifications. We are, however, confident that both demerits can be overcome in future by automated constraint pre-processing. There are, e.g., techniques for automated miniscoping [de 86].

Part V

Conclusions

Chapter 14

Related Work

Before we discuss related work, we summarize the most relevant properties that, put together, distinguish our approach:

- Our system has a coherently formal underpinning. This includes object types, a notion of object content, system states, state changes, preservation requirements, and a functional language for implementing migration algorithms.
- Despite the formal underpinning, our system is fully implemented.
- Our approach is declarative — we abstract from concrete transformation approaches.
- We sacrifice decidability for expressiveness; concept and requirements specification base on full First Order Logic.
- Objects are immutable; new versions can be created by transformations only. This property is provably guaranteed by our system.
- Our approach is independent from particular object formats and technologies for describing such (like XML Schema, XML DTDs). Also, we support user-defined functions and predicates.
- We trace object histories using a well-defined transformation operation; object histories need not be part of the model for digital archives.
- Preservation requirements relate to object histories, concepts, and contexts. Concepts are used like predicates and, thus, hide their implementation; this supports re-use and readability.

In the following sections we discuss related work in detail.

14.1 Migration in Digital Archives

The Reference Model for Open Archival Information Systems (OAIS, [Con02]) is widely accepted nowadays as a reference model for Digital Archives. It introduces basic terminology, provides an abstract information model for digital objects, includes an organizational model for DAs, defines a basic functional model for processes that run inside DAs, and describes concrete preservation methods like migration. These aspects bear general conditions of the application domain that drove our research. In particular, this

reference model can be used to classify our approach: We support *Preservation Planning* in *expressing* their preservation requirements and provide automated support for the *Administration* in checking whether the migration results meet these requirements. There, we can handle *content* migration and partly support model construction.

In [SRR+06, SBNR07] a test bed for evaluating preservation strategies is introduced. It defines an environment for repeatable experiments in standardized laboratory settings. This particularly includes a workflow of how to conduct experiments. This test bed underlines the necessity for a well-founded methodology for evaluating preservation procedures. Our methodological approach to the case study has been influenced by the workflow introduced in [SRR+06]. Also, the preservation-centric view and integration into existing environments are design goals that go back to these works. Apart from that, earlier case studies carried out using the test bed have shown that preserving text-based formats like e-mail can be handled well by migration [Dig01]. In contrast, preserving dynamic media is stated to be better-handled by emulation; this drove our decision to exclude dynamic documents. We are convinced that the formal underpinning and our preservation language are well-suited for being integrated into this test bed.

14.2 Migration and Transformation in Other Contexts

As *model transformation* has applications in many other areas like model-driven software development (MDD) or automated knowledge exchange [FP00, RSV04, SKB06, PHS+06], many approaches exist in this respect. When applied, transformations usually are designed to *preserve* certain properties. In MDD, e.g., abstract models are transformed into more specific ones while preserving the behavior of the specified software system [MCG05].

QVT [Obj05] is an OMG standard and defines a standard way to transform models that conform to given MOF-based meta models. There, the QVT language integrates the OCL 2.0 standard [GRB03, Ric02] for selecting model entities. Apart from included model mapping languages, QVT can be used declaratively and supports the so-called QVT/BlackBox mechanism. The latter can be used to invoke transformations that are expressed in other languages like XSLT or XQuery. For each transformation step, QVT generates a so-called *trace instance*; tracing is supported by this system and can be used, e.g., for model synchronization. Hence, we share some important concepts with QVT: (1) The declarative style, (2) independence of concrete transformation approaches, and (3) facilities for object tracing. Our system, however, includes a notion of object contents and a preservation language. We can check object traces w.r.t. given preservation requirements. Also, our approach is MOF-independent.

Database migration is a widely examined research field; a literature research has been conducted in [Bar04]. In [MDM+94] technologies and methods are discussed for handling hierarchically structured data by relational databases; [AFL02, BFM05] deal with XML serialization of database content. As XML serialization is an important approach to preserving databases, this can directly be transferred to our application domain. Both applications particularly show that migrations require for approaches that do not rely on specific technologies for representing digital objects. We argue that our ADT-based approach offers sufficient flexibility here. Apart from that, schema transformation ap-

proaches are widely practiced in database migration as well. Usually the physical data itself is not transformed, but queries are translated from the new schema to the old one. *Reversible* transformations and *decidability* of the underlying transformation approach are of major interest [MP99, Hai05]. The former allows for bi-directional mappings; the latter assures automation. Although reversible transformations are desirable in our setting as well [GW05, Con02], they cannot be expected in large-scale projects like document format transformations. However, our state-based approach naturally allows for rollbacks if the original objects are kept. Both together drove our decision to sacrifice decidability for expressiveness. We do not primarily aim at *generating* transformations but focus on *verifying* adherence to given preservation requirements.

Although our approach is independent of a particular transformation approach, some basic concepts have been adapted from application-independent methodologies that exist in this area. First, diverse variants of graph transformation [Grz97, SWZ99, EGdL+05] exist for automating model transformations. In particular, approaches like [SWZ99] offer control mechanisms. This results in executable specifications. In [KK96] a unified theory for graph transformation is introduced. Basically, matches are comparable to concept matches in our setting. We adapt the basic idea to match the left-hand side of a "rule" (concept) in the source context and integrate the right-hand side into a target context. However, we use concepts as a means of abstraction. Concept interfaces allow for "selective" tracing and do not cover the whole context. Also, we use an explicit transformation operation and embed migrations into an archival context. In [KK96], transformations are modeled by rule application. Although control mechanisms like "iterated application" guide the transformation process, explicit tracing is not supported. However, integrating control features may be beneficial for our approach as well. There, theoretical approaches of [KK96] like the "interleaving semantics" of graph transformations are interesting for future work. Object tracing in conjunction with graph transformation is examined in [MDJ02, MEDJ05]. There, Mens et al. use *tracking functions* in the context of behavior preserving model refactorings. Tracking functions relate vertices of source and target graphs. That way sequential rule applications can be traced back and preservation-oriented proofs can be conducted. We have adapted this technique to our needs; traces relate interface objects that are subject to preservation constraints. Yet we use an explicit transformation operation whereas Mens et al. annotate graph rewriting rules by tracking functions.

Second, ontology mapping approaches [KS03, SM01, ES07] have been subject to increasing interest over the past decade. This resulted in growing tool support [GYS07, LS06]. Early applications comprise knowledge exchange and database migration. In the meantime, approaches like [HC04, FBR06] have been used in digital archiving as well to detect automatically when objects are outdated and to propose appropriate preservation strategies. We particularly adapt the notion of a *concept*, which describes semantic relationships in ontologies as well. However, ontology mapping is usually based on *decidable* description logics so as to support automation. We, however, need more expressive logics in order to describe sophisticated document formats (like XML Schema [Wor01]). Apart from that, the constructor-based approach of ADTs is more suitable for tree- or graph-based formats which frequently occur in our domain. However, ontologies are surely interesting during the concept identification phase of our approach. Also,

exchanging concept specifications between different archives may require for concept mappings; in different archives, different concept hierarchies may have evolved that cover similar properties. These considerations open interesting topics for future research.

14.3 Notions of Preservation

Formal notions of preservation can be found in different variations throughout the relevant literature [MCG05, Por05, Grz97, WO00, ADK06]. Refinement, e.g., usually means a reformulation of system properties in the same language such that the newly specified system inherits all properties of the former system [MCG05]. The *Typed Object Model* (TOM, [WO00]), e.g., fully bases the notion of object preservation on refinement. The system is discussed in more detail in Sect. 14.5. Although we explicitly support refinement using formal ADTs, basing preservation purely on refinement seems to be too restrictive. We rather prefer to keep the degree of formality (trustworthiness) scalable.

In node replacement approaches of graph transformation, authors usually speak of preservation w.r.t. those nodes and edges of the source graph that are not affected by a rule application [Grz97]. However, usually node/edge *identity* is required, which is too restrictive for our purposes. As has been shown in [MEDJ05], explicit object tracing is required in order to reason about "functional" preservation.

In [ADK06] preservation of FOPL formulas under model homomorphisms is studied with applications to database queries. In principle, we use homomorphisms as well; source and target models are abstracted to concept interfaces. The semantics of object abstraction, however, is user-defined in our system. In particular, abstraction functions can be specified formally. We favor this approach as it does not introduce an inherent formal notion of preservation; the user has to implement "preservation" himself while we check this property w.r.t. different versions of given objects.

Semantic values have been introduced in [SSR94]. They are to enable interoperability between different database systems. A semantic value is a concrete value together with a context. Contexts describe a set of properties that apply to that value and, thus, can be understood as meta data. They guide *transformers* that are used to interpret semantic values in concrete systems. As an example, the value 1.25 together with the context $Currency = US - Dollar$ forms a semantic value. It may be interpreted in countries other than the US by transforming it to the local currency. We have adapted the basic idea to permit different representations for semantic objects. Semantic values can be compared to digital objects in our system. In both approaches transformations ("conversions" [SSR94]) have to preserve *semantics*. In [SSR94] source and target object have to be comparable w.r.t. a reference context. This very much corresponds to the notion of preservation in TOM (type abstraction) and can be implemented in our system as well. The approach in [SSR94], however, does not support semantic relationships and tracing.

14.4 Formal Approaches to Digital Archiving

In [CLB01] the authors concentrate on preservation *strategies*. Archives are described by algebras. Objects themselves are modeled as state sequences representing the object's

history. An object is preserved if it has the same interpretation in all states of its life cycle. Game theory is employed to model an indifferent two-player-game of the archive against environmental influences. Both sides can take specific actions (e.g., migration on the archive's side and data corruption on the environment's side). The mutual effects then can be studied, which results in suitability assessments of certain preservation strategies. We have adopted the idea of describing archives by algebras. However, we concentrate on the *operational* impact of migrations; we do not consider strategies. Object representations may change. Therefore, we support different implementations of one and the same aspect. Also, we cover object relationships, which play only a minor role in [CLB01], quite extensively. Apart from that, we have a *generic* notion of preservation; users have to *implement* it.

5SL [GF02] is a formal description language for specifying digital libraries. It bases on the 5S-theory [GFWK04] and allows for automated derivation of prototypes that meet a given specification. Object contents is described by streams; structural information is described by structures. Spaces model user interfaces. Internal processes like searching are defined via scenarios. Societies model communities that access digital libraries. The whole theory does not so much address preservation and migration as the interaction of digital libraries, their internal processes, and user access via interfaces. Hence, the 5S theory can be used to describe the *environment* in which our approach is integrated. In particular, we see interesting points for future research concerning descriptions of internal processes.

14.5 Systems for Formal Quality Assurance

The Typed Object Model (TOM, [Ock98]) incorporates and deploys type descriptions via a type brokerage system. It focuses on file format transformations and covers structured file format transformations with focus on preservation. The system is object-oriented and administrates type hierarchies with formal pre-and post conditions and type invariants. Preservation is defined w.r.t. sub-typing. Object preservation means preserving object content w.r.t. a common super-type of the source and target object; sub-typing is used as abstraction function. We borrow the idea from TOM to base our notions for object preservation (functional concepts) on object types and abstraction but implement it in a less restrictive way. Also, TOM does neither incorporate object relationships (non-functional concepts) explicitly nor an archival context. In particular, it has no formal notion of transformation and does not support object tracing.

The xlinkit system [NCEF02] deals with automatic checks of semantic document properties. It allows for expressing properties of interrelated documents by means of FOPL-formulas. Documents and their content may be addressed by XPath-expressions. These XPath expressions are then used to generate "smart" links between those documents that violate given consistency requirements.

In [SBRS04, Sch04] this idea has been extended. Document repositories are examined with particular focus on version control systems. Documents are defined via a language of first order *temporal* logic. Thus, consistency rules between documents in different system states can be specified. A provably correct-working, iterative consistency checking strategy is developed that pinpoints inconsistencies and suggest repair actions.

Both, the xlinkit and the CDET system, are fully implemented and have been evaluated in practice. As both systems use FOPL formulas to describe document properties we have incorporated FOPL as well. The xlinkit system, however, does not incorporate transformations at all. Consistency conditions are checked w.r.t. static document sets. Document versioning in CDET can be understood as a migration; document properties before and after a state change can be related using temporal logics. We share some basic ideas. First, we use types and sub-typing to structure objects and object contents. Second, both systems support object histories and have a formal underpinning. Third, we specify requirements ("consistency rules" in CDET) using temporal logics. In CDET, however, branching is not supported. Also, we implement a preservation-oriented view; we use object traces to relate source and target objects directly w.r.t. concept interfaces. There, concepts provide abstraction. Preservation requirements have to refer to a concept but need not refer to concrete implementations. In contrast, consistency rules in [SBRS04] exhibit all details. Also, the approach lacks a suitable notion of preservation. We argue that our preservation constraints are easier to communicate to archivists than temporal formulas.

14.6 Graph-based Queries

Automated link generation has been examined in the digital library community for quite a while. There, techniques are studied for using information retrieval to generate "semantic" hypertext [ACM97, Gol97, WS99]. Continuative approaches deal with translating user queries to corresponding hypertext automatically [HL07]. Automated link generation is understood as a means for customizing search results and representing them as hypertext in a convenient manner [OC03]; computer-human interaction approaches come into play. We, however, focus on techniques for generating well-formed and valid links. This includes link translation during transformation processes in order to preserve link consistency.

As mentioned before, the xlinkit system [NCEF02] incorporates XPath expressions for addressing documents. In particular, the system generates navigable consistency reports. This feature supports user-acceptance and helps to resolve inconsistencies in interrelated documents. Although the need for a general integration mechanism for query languages has been identified, such a mechanism has not yet been provided. The system rather fixes the XPath semantics statically. Due to the practical experiences made with xlinkit and due to our experiences with html linking, we have found integrating query languages important for gaining practical acceptance. Although it requires some effort in advance (graph specification scheme, transformation of context-free grammars), we argue that the benefits overweigh.

Due to the rapid emergence of the XML technology, there is much related work available dealing with query languages based on tree automata. Usually, a tree automaton [CDG$^+$07] is constructed for a given query; the set of *all* accepted trees is considered the semantics of that query. This approach targets at drawing conclusions w.r.t. query subsumption and is well-suited for that purpose. In our scenario, however, this technique requires for constructing tree automata for every single query so as to check whether the automaton accepts the given structure. This procedure tends to be time-consuming

and usually is not needed; we do not want to reason over query subsumption properties. Instead we need a trustworthy mechanism to semantically evaluating *and* constructing queries. Apart from that, tree automata based techniques embody parts of the query semantics already in their state transitions [Nev02, Chi00, BALD03]. In order to capture this semantics, the *user* would have to specify the tree automata in question. This is impractical in our setting since it has to be done for every single query. In contrast, our approach implements a mechanism that accepts all queries of a given language (due to its formal grammar) and computes semantic query results when applied to given query structures.

In general, applying product automata for dealing with syntax and semantics in parallel is a standard technique in the literature. In model checking, e.g., software systems and formulas of temporal logics are both translated to Finite State Automata [HMU06, MOSS99, RSV04]. We have, however, adjusted these standard techniques such that they are applicable conveniently in our setting. In particular, we do not translate single queries (as is done in model checking) due to the reasons explained above. Also, we use hierarchical automata in order to keep the number of states to a minimum — state space explosion is a well-known problem in model checking and, in general, whenever product automata are constructed.

Chapter 15

Conclusion and Outlook

15.1 Summary

In Sect. 1.3 we have formulated our overall objective as follows:

> The results of this thesis contribute to a better automation of quality assurance for migration processes in digital archiving. Basing on suitable formal notions of information and preservation, our method allows to *express* preservation requirements formally and *evaluate* adherence to them in an automated way. The formal underpinning of our approach results in a high degree of trustworthiness. Practicability, well-foundedness, the ability to *pinpoint* violations of preservation requirements and smooth integration into internal workflows of DAs have been important design goals for our method.

We sum up by showing how our approach meets these functional and non-functional requirements.

Improve Automation in Quality Assurance

Our framework allows to formulate preservation requirements formally. When executing migrations, our prototype implementation traces changes to digital objects automatically. Also, violations of given requirements are pinpointed. Apart from that, we have demonstrated model construction facilities in the case study. URLs could be transformed automatically such that they met the underlying preservation requirements.

Both applications support automation of quality assurance. The former helps in verifying the output of migration algorithms and detecting functional errors. The latter (partly) generates correct-working migrations. As models grow large, automation becomes vital. Therefore, we consider our contributions to be of high relevance. Full automation, however, cannot be expected in general. Some properties cannot be handled fully formally, others are not worth the effort.

Suitable Notion of Preservation

Our notion of preservation relates source objects and target objects. A property is preserved if the target objects are new versions of the source objects and the property

equally holds for both the source and target objects; preservation considers object histories (equivalently: object traces). However, different implementations of the property may be used for the source and target objects; we support *content migration* ([Con02]).

In our system, we use the term "concept" instead of "property"; this stresses connections to concepts in ontologies. Apart from their implementations, concepts include a well-defined interface for matching. As we distinguish functional and non-functional concepts, our notion of preservation applies to both object contents and object relationships. On the whole, it implements the view on "preservation" in digital archiving directly (cf. [Con02, Tas96]); we conclude that it suits the needs in this domain. This has been proved by our case study where we formulate and verify relevant preservation tasks related to a website migration.

Expressing Preservation Requirements

We use an expressive *preservation language*. Preservation constraints are at the heart of this language. Their semantics directly implements the above-described view on preservation. Hence, they contain source objects, a concept, a context for matching the source objects and a context for matching the target objects. There, we use a compact notation in order to keep specifications short and readable. A wildcard can be used instead of concrete source and target contexts. That way we support evolving concept specifications. This is particularly important in digital archiving. As technologies evolve, concepts may be extended by new implementations.

Preservation formulas express more sophisticated preservation requirements. They can be used to select object collections and to apply preservation constraints to all contained objects. Using trace quantifiers we support branching object histories. They specify whether a constraint must apply to all traces of the source objects or to at least one trace. On the whole, our preservation language bases on first order logic. It offers sufficient expressive power for relevant practical examples. However, the language can be extended or customized to more specific needs.

Practical Applicability

Our system is fully implemented and supports all steps that are necessary to formulate and evaluate preservation requirements. Also, it meets basic requirements of digital archiving:

- We are independent of particular object formats and technologies for describing such (like XML Schema, XML DTDs).
- We support user-defined functionality. Implementations can be integrated into our prototype system and evolve over time.
- Objects are immutable. The formal underpinning of our system guarantees this property. For this purpose, we have introduced an explicit notion of object contents.
- Preservation requirements are formulated from an archivist's perspective as described above; as preservation constraints refer to concepts and hide implementations, specifications remain "readable".

- Migration algorithms can be implemented using our functional language. This language is integrated into the overall framework and allows for automated change tracking.

The case study has shown applicability of our methods; we could handle complex properties in an automated way. Performance measurements have shown acceptable results for model sizes of practical relevance.

Well-foundedness

All parts of our system have been specified using formal methods. In particular, we have introduced digital archives together with a notion of a "permissible extension". Also, we have introduced a formal notion of object content. Well-defined state transitions can change an archival states only. If a digital archive has been extended in a "permissible" way, these state changes provably guarantee object immutability and uniqueness of object IDs. Even if an object is deleted, its ID will never be used again.

Apart from that, concepts, preservation formulas, and our functional language have a well-defined semantics. In all that, we have used standard methods like Abstract Data Types (ADTs) and Abstract State Machines (ASMs), which are well-studied. Together with the coherently formal underpinning our approach is a step towards a high degree of "trustworthiness".

Pinpoint Constraint Violations

Preservation requirements are defined for concrete objects. As we trace object histories, constraint violations can be reported for concrete source objects, target objects, and w.r.t. a concept. This is vital when interpreting unexpected migration results. It may particularly help when developing migration algorithms; this is a lesson we have learned from our case study.

Work Flow Integration

We presuppose little system knowledge only. Objects must have an ID, which is unique system-wide, and the sets of existing objects and used IDs have to be known. As these requirements are a must for digital archives, our approach can be applied in that domain. Our basic digital archive can be extended and customized by user-defined datatypes and functionality. Also, we are independent of a particular transformation approach. Users can choose whether they use our programming language or implement their migrations in other languages and connect them to our system. When using our functional language, we permit full access to all parts of a digital archive. This particularly includes the set of existing objects and object IDs. That way *internal* migration processes can be implemented; no intermediate access control is established.

15.2 Future Work

On the whole, we conclude that this thesis meets our initial claims; we have defined and implemented a framework for supporting automated quality assurance of migration processes in digital archiving. However, important points for future extensions remain.

In particular, further case studies need to be conducted. There, we do not focus on digital archiving only; other domains like model-driven software development seem to offer interesting applications as well. Additional case studies will help to develop domain-specific procedure models. Also, they will suggest best practices in embedding our framework into existing workflows. Concerning digital archiving, integrating our approach into the test bed developed in [SRR+06, SBNR07] seems useful. Apart from that, future work should cover the following areas.

Functional Extensions

Some functional limitations of our approach have been mentioned previously; future work should examine how they can be overcome. Future extensions should particularly cover:

Dynamic content:
: Dynamic object content appears in many relevant applications. Web forms and video/audio streams, e.g., naturally occur when dealing with website preservation. Formal handling of video and audio seems difficult; both contain continuous data and require a hardware/software environment for rendering. Discrete systems, however, may be handled by automata-based techniques; interaction using web-forms is an example. User-support in modeling dynamic content is desirable here.

Deferred migration:
: Migrations may be expensive and induce risks to digital objects. It may, thus, be sensitive to defer them until triggers like “access” or a specific date activate it. A possible solution could be to mark datatypes as “deprecated”; access to an object of a deprecated type then requires a previous transformation. Integrating a related mechanism into our approach seems to be useful. Relevant technologies exist in the database community, for example.

Graph-based queries:
: We support regular query languages only. Standard languages like XPath and CSS-selectors, however, additionally support predicates and terms for selecting sets of tree nodes. Since we integrate user-implemented functions, integrating the full language into our automata-based techniques seems feasible. Special “evaluation nodes” or delay nodes as proposed in [Mai06] could be used.

Migration algorithms:
: Our functional language does not support pattern matching and higher order functions. The former supports readability, the latter supports abstraction and re-use. Both extensions are standard in functional programming [Tho99, Jon95, GPN96, Erw96] and may be implemented with moderate effort.

Improvements Through Formal Techniques

Structuring concepts:

We do not yet include structuring and re-use mechanisms for concepts. In particular, concept interfaces are generic in the object types only; generic predicates or functions are not supported. Also, concept implementations cannot include concept formulas. Integrating these features requires some technical effort but is useful. First, proofs can be structured and re-used. Second, concept libraries and concept hierarchies can be set up. The CASL language semantics [CoF04] may serve as a guide when integrating these features; most of them are supported by the CASL language.

Reasoning support:

As we use First Order Logic, the implication problem is undecidable, in general. Nevertheless, many applications still allow for formal reasoning. Also, user interaction can guide the inference process. Formal reasoning support may be useful for the following areas, where integration with existing theories and theorem provers ([Isa02, McC06, McC03]) should be examined as well:

Concepts:

Detecting implications among (*static*) concept specifications helps to reduce redundancy. This speeds up the process of checking preservation requirements. Suitable techniques exist (e.g., [CoF04]); integrating them seems to be a matter of implementation.

Truth values of concept formulas may change in the course of system *evolution*. In order to avoid unnecessary concept re-evaluations, a theory may help in characterizing those state changes that preserve truth values of concept formulas.

Preservation:

Similar to concepts it is desirable to detect redundancies among preservation requirements. A theory for reasoning about preservation formulas has to include object histories, concepts, and system states. Existing proof theories for temporal and model logics may serve as an orientation here.

Migration Algorithms:

Reasoning support for functional programs is a well-established research field. However, relevant approaches focus on term equality, which is of minor interest in our setting; our programs produce migration sequences and, hence, side effects. It would be useful to have a theory that allows for deriving properties like "Function f satisfies preservation formula ϕ". This theory would, thus, integrate functional programs, object histories, and preservation formulas. To our appraisal, it is likely that existing theories for reasoning about functions with side-effects (e.g., [Tal98]) can be customized.

Model coverage:

In our case study several case distinctions had to be made when specifying directory structure preservation. As applications get complex, this may lead to "gaps" in model coverage. These, in turn, may cause unintended migration effects to go

unnoticed. Formal approaches to test coverage (e.g., [HLSU02]) may be interesting here.

Ontologies:
Digital archives usually use standard formats for their hosted digital objects. Also, related formats usually support related concepts; re-using concept specifications should be supported. Concepts and their interrelations are often described by ontologies. Ontologies may, thus, serve as an orientation when identifying relevant concepts that are subject to preservation. Also, specifications may be subject to formal proofs whether they match a given ontological description. Different communities, however, may use different terms for describing equivalent aspects. This may prevent re-use. Therefore, integrating our system and approaches to ontology merging (e.g., [KS03]) seems to be beneficial.

Constraint evaluation:
We have shown that the way of formulating preservation requirements considerably influences evaluation performance. Minscoping [de 86, Wan60] has helped to decrease runtimes; other techniques related to static pre-processing should be studied as well.

In [Sch04] FOPL formulas are evaluated iteratively in an evolving system. It has been shown that this results in considerable performance improvements. As preservation formulas base on first order logic, the approach seems to be applicable to our system.

Implementation

We have developed a prototype in order to evaluate our methods. It can be understood as the core of a more sophisticated system that can be

- used as a testing environment or
- integrated into real-life systems.

The following improvements will serve these purposes and potentially increase user-acceptance:

Language interfaces:
Up to now user-defined data types, functions, and predicates have to be implemented in JAVA. If our system, however, is to be included in existing software environments, interfacing with other languages like C and C++ may be required. The current implementation has already been designed in view of these extensions; syntax and semantics are handled fully independently of each other.

User interfaces:
Due to the formal underpinning, strong expert skills are necessary to apply our methods. A coherent IDE supporting datatype, concept, and constraint specification as well as programming migration algorithms will accelerate the overall process. Also, visualization techniques may increase user-acceptance. Constraint coverage or constraint violations, e.g., could be displayed w.r.t. concrete models; this can help experts in communicating these aspects to non-experts.

Code generation:

State-of-the-art CASE tools offer code generation facilities; otherwise practical acceptance would suffer. Apart from the standard translation of datatypes with attributes and constructors into code frames, we see the following potential:

Graph-based querying:

Our prototype already includes a framework for handling automata. The graph specification scheme introduced in this thesis allows for code generation. The resulting automata *refine* general hierarchical automata by constraining the sets of permissible states and state transitions. Indeed, we have generated the code for the running example. As our prototype is implemented in JAVA, refinement has been realized by sub-classing and method overriding.

Migration algorithms:

Instead of compiling our functional language to machine code, it seems useful to offer code translations for the most common programming languages like C, C++, JAVA etc. Migration algorithms, thus, could be translated into the language that is used for datatype, function, and predicate implementations. The overall system then can be compiled with state-of-the-art compilers. They are usually highly optimized and produce very efficient code.

List of Figures

List of Tables

Appendix A

Specification of the Basic DA

The sentences Sen_s^{DA} of the specification of the basic digital archive are determined by the following data type specifications.

$$
\begin{array}{l}
\texttt{type Bool}<\{\texttt{Top}\} = \\
\texttt{begin} \\
\quad \texttt{constr} \left[\begin{array}{l} \texttt{True} \quad : @ \\ \texttt{False} \ : @ \to @ \end{array} \right. \\
\quad \texttt{ops} \left[\begin{array}{l} not \quad : @ \to @ \\ and \quad : @ \times @ \to @ \end{array} \right. \\
\quad \texttt{axioms} \left[\begin{array}{l} not(\texttt{True}) = \texttt{False} \wedge not(\texttt{False}) = \texttt{True} \wedge \\ and(\texttt{True}, \texttt{True}) = \texttt{True} \wedge and(\texttt{True}, \texttt{False}) = \texttt{False} \wedge \\ and(\texttt{False}, \texttt{True}) = \texttt{False} \wedge and(\texttt{False}, \texttt{False}) = \texttt{False} \end{array} \right. \\
\texttt{end Bool};
\end{array}
$$

$$
\begin{array}{l}
\texttt{type OID}<\{\texttt{Top}\} = \\
\texttt{begin} \\
\quad \texttt{constr} \left[\begin{array}{l} \texttt{initID} \ : @ \\ \texttt{nextID} \ : @ \to @ \end{array} \right. \\
\quad \texttt{preds} \left[\begin{array}{l} <_{id} \ : @ \times @ \end{array} \right. \\
\quad \texttt{ops} \quad [\\
\quad \texttt{axioms} \left[\begin{array}{l} \forall id : @ \ \bullet \ \texttt{initID} \neq \texttt{nextID}(id) \\ \forall id, id' : @ \bullet \ <_{id} (id, \texttt{nextID}(id)) \\ \forall id, id', id'' : @ \bullet \ <_{id} (id, id') \wedge <_{id} (id', id'') \Rightarrow <_{id} (id, id'') \\ \forall id : @ \ \bullet \ \neg <_{id} (id, id) \end{array} \right. \\
\texttt{end OID};
\end{array}
$$

$$
\begin{array}{l}
\texttt{type DObj}<\{\texttt{Top}\} = \\
\texttt{begin} \\
\quad \texttt{constr} \left[\begin{array}{l} \texttt{createDObj} : \texttt{OID} \to @ \end{array} \right. \\
\quad \texttt{ops} \quad \left[\begin{array}{l} \texttt{oid} : @ \to \texttt{OID} \end{array} \right. \\
\quad \texttt{axioms} \left[\begin{array}{l} \forall id : \texttt{OID} \ \bullet \ \texttt{oid}(\texttt{createDObj}(id)) = id \end{array} \right.
\end{array}
$$

end DObj;

type Set[Top]<{Top} =
begin

constr	$\{\}$	$: @$
	$\{\}_s$	$: \texttt{Top} \to @$
	$\cup$	$: @ \times @ \to @$ (assoc,comm,idem,unit $\{\}$)
preds	*empty*	$: @$
	$\in$	$: \texttt{Top} \times @$
	$\subseteq$	$: @ \times @$
ops	rep	$: @ \to \texttt{Top}$
axioms	$\forall s : \texttt{Set[Top]} \bullet \mathit{empty}(s) \Leftrightarrow s = \{\}$	
	$\forall x : \texttt{Top}, s : \texttt{Set[Top]} \bullet \neg x \in \{\}$	
	$\forall x : \texttt{Top}, y : \texttt{Top}, s : \texttt{Set[Top]} \bullet x \in \{y\}_s \Leftrightarrow x = y$	
	$\forall x : \texttt{Top}, s : \texttt{Set[Top]}, t : \texttt{Set[Top]} \bullet x \in s \cup t \Leftrightarrow x \in s \vee x \in t$	
	$\forall s : \texttt{Set[Top]} \bullet \{\} \subseteq s$	
	$\forall x : \texttt{Top}, s : \texttt{Set[Top]} \bullet \{x\}_s \subseteq s \Leftrightarrow x \in s$	
	$\forall s : \texttt{Set[Top]}, t : \texttt{Set[Top]}, u : \texttt{Set[Top]} \bullet s \cup t \subseteq u \Leftrightarrow s \subseteq u \wedge t \subseteq u$	
	$\forall s, s' : \texttt{Set[Top]} \bullet s = s' \Leftrightarrow (\forall x : \texttt{Top} \bullet x \in s \Leftrightarrow x \in s')$	
	$\neg \texttt{inst}_{\texttt{Top}}(\texttt{rep}(\{\}))$	
	$\forall s : \texttt{Set[Top]} \bullet s \neq \{\} \Rightarrow \texttt{rep}(s) \in s$	
	$\forall s : \texttt{Set[Top]}, x : \texttt{DObj} \bullet x = \texttt{rep}(s) \Rightarrow$	
	$\quad \forall y : \texttt{DObj} \bullet y \in s \Rightarrow (\texttt{oid}(x) = \texttt{oid}(y) \vee <_{id} (\texttt{oid}(x), \texttt{oid}(y)))$	

end Set[Top];

The annotation "(assoc, comm,idem,unit {})" is adopted from the CASL language and states that $\cup$ is associative, commutative, idempotent, and has unit {}.

Appendix B

Continuative Examples on Formal Parts

B.1 Objects and Digital Archives

In the following we provide examples on well-typedness of terms, on the basic DA, and on migration sequences (Chap. 3).

Sample Derivations for Terms and Subterms

We derive well-typedness of $\texttt{name}(\texttt{home}(x)) \in T_{\texttt{String}}(X, \Sigma)$, where $X = \{x : \texttt{Website}\}$ and

$$\begin{aligned}\Sigma = (&\{\texttt{Website}, \texttt{HTMLDoc}, \texttt{Doc}, \texttt{String}\},\\ &\{\texttt{Website}{<}\texttt{Website}, \texttt{HTMLDoc}{<}\texttt{HTMLDoc}, \texttt{HTMLDoc}{<}\texttt{Doc}, \texttt{String}{<}\texttt{String}\}\\ &\emptyset,\\ &\{...\}\\ &\{\texttt{name} : \texttt{Doc} \to \texttt{String}, \texttt{home} : \texttt{Website} \to \texttt{HTMLDoc}, ...\}).\end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{x : \texttt{Website} \in X}{x \in T_{\texttt{Website}}(X, \Sigma)} \quad \dfrac{\texttt{home} : \texttt{Website} \to \texttt{HTMLDoc} \in \mathcal{F}}{}}{\texttt{home}(x) \in T_{\texttt{HTMLDoc}}(X, \Sigma)} \quad \texttt{HTMLDoc}{<}\texttt{Doc}}{\texttt{home}(x) \in T_{\texttt{Doc}}(X, \Sigma)} \quad \texttt{name} : \texttt{Doc} \to \texttt{String} \in \mathcal{F}}{\texttt{name}(\texttt{home}(x)) \in T_{\texttt{String}}(X, \Sigma)}$$

In order to have an example for subterm relationships, we provide the derivation for $x \in T^{\leq}_{\texttt{Website}}(X, \texttt{name}(\texttt{home}(x)))$.

$$\dfrac{\dfrac{\dfrac{\texttt{home}(x) \in T_{\texttt{HTMLDoc}}(X, \Sigma)}{x \in T_{\texttt{Website}}(X, \Sigma)}}{x \in T^{\leq}_{\texttt{Website}}(X, \texttt{home}(x))} \quad \dfrac{\dfrac{\texttt{name}(\texttt{home}(x)) \in T_{\texttt{String}}(X, \Sigma)}{\texttt{home}(x) \in T_{\texttt{HTMLDoc}}(X, \Sigma)}}{\texttt{home}(x) \in T^{\leq}_{\texttt{HTMLDoc}}(X, \texttt{name}(\texttt{home}(x)))}}{x \in T^{\leq}_{\texttt{Website}}(X, \texttt{name}(\texttt{home}(x)))}$$

The assumption $\texttt{home}(x) \in T_{\texttt{HTMLDoc}}(X, \Sigma)$ can be derived as shown above.

Permissible Archive Extensions

Simple extensions of type `Dir` may serve as examples for the ID and constructor properties.

ID-property: Recall the record-like type `Dir` (Fig. 3.1, page 28). It has a constructor

$$\texttt{Dir} : \texttt{OID} \times \texttt{String} \times \texttt{Set}[\texttt{Dir}] \times \texttt{Set}[\texttt{Doc}] \to \texttt{Dir}$$

and axioms of the form

$$\forall id : \texttt{OID}, n : \texttt{String},\\ subDirs : \texttt{Set}[\texttt{Dir}], subDocs : \texttt{Set}[\texttt{Doc}] \bullet a(\texttt{Website}(id, n, d, h)) = \_$$

for all attributes a (including `oid`). There, _ stands for the constructor parameter that corresponds to a. It is easy to see that datatypes of this style satisfy the ID-property.

Constructor-property (1): All record-like data types also satisfy the first part of the constructor property. Their attributes simply return the constructor parameters.

As a counterexample, suppose we had an additional constructor

$$\texttt{Dir2} : \texttt{OID} \to \texttt{Dir}$$

and

$$\texttt{rep}(\texttt{srcDirs}(\texttt{Website2}(id))) = \texttt{Dir}(\texttt{nextID}(id), \text{“}\texttt{some}\text{“}, \emptyset, \emptyset).$$

Then $\texttt{rep}(\texttt{srcDirs}(\texttt{Website2}(id)))$ does not contain $\texttt{Dir}(id, \text{“}\texttt{some}\text{“}, \emptyset, \emptyset)$. This violates the constructor property. Intuitively, constructor property (1) requires that all digital objects that can be extracted from another object have to be provided upon creation of that object already; we prohibit “default values” that have an object type. Only this facilitates to characterize object-valued content by *constructor terms*.

Constructor-property (2): If type `Dir` consists of the constructors `Dir` and `Dir2`, only, the second part of the constructor property is satisfied. Even if there were values that can be constructed by both constructors, a constructor term using `Dir2` is truly shorter then a term using `Dir`. Thus, the property remains valid. In general, this part of the constructor property assures that the object-valued content can be *uniquely* determined by using *minimal* constructor terms.

Object-Valued Content

Recall the constructor term

$$x_3 := \texttt{Dir}(\texttt{nextID}^3(\texttt{initID}), \text{“}\texttt{source}\text{“}, \{x_1, x_2\}, \{x_0\})$$

representing the directory “source” (cf. Tab. 3.1, page 30), where we abbreviate

$$x_0 := \texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, ...),\\ x_1 := \texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\}),\\ x_2 := \texttt{Dir}(\texttt{nextID}^2(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{\}).$$

Since types `Dir` and `HTMLDoc` have one constructor only, x_3 is at the same time minimal and unique for the represented value. According to Defn. 3.3.3 $cont^{\mathcal{A}}_{\texttt{DObj}}(x_3)$ yields the set of all sub-terms of x_3 of type `DObj`. Hence, $cont^{\mathcal{A}}_{\texttt{DObj}}(x_3) = \{x_0, x_1, x_2, x_3\}$.

Also, this confirms reflexivity of the relation $R^{cont}_{\mathcal{A}}$ in Lemma 3.3.3. Generally, the sub-object-relationship is an ordering and, hence, can be represented by Hasse-diagrams. In particular, the graphical visualizations for directory and file structures that we have been using so far represent sub-object relationships.

As another example, the term $\{\} \cup \{x_3\}_s$ represents a set containing the directory "source". This term is not minimal. The same set can be represented by $\{x_3\}_s$. Hence, the set of all subterms of type `DObj` of $\{x_3\}_s$ determines the object-valued content of $\{\} \cup \{x_3\}_s$, i.e., $cont^{\mathcal{A}}_{\texttt{DObj}}(\{\} \cup \{x_3\}_s) := \{x_0, x_1, x_2, x_3\}$.

The term $\texttt{rep}(\{x_3\}_s)$ chooses a representative of $\{x_3\}_s$. Since `rep` is no constructor, this term meets the prerequisites that are necessary for Lemma 3.3.3(3) to be applicable. According to this lemma, the result of $\texttt{rep}(\{x_3\}_s)$ is in $cont^{\mathcal{A}}_{\texttt{DObj}}(\{x_3\}_s)$. Indeed, $\texttt{rep}(\{x_3\}_s) = x_3 \in cont^{\mathcal{A}}_{\texttt{DObj}}(\{x_3\}_s)$.

Migration Sequences

The example trace of Tab. 3.1 on page 30 is represented by the migration sequence

$$\begin{aligned}\Delta := \langle\, & \mathcal{A}_0,\ \mathsf{cre}(\texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, ..., \emptyset)),\\ & \mathcal{A}_1,\ \mathsf{cre}(\texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\})),\\ & \mathcal{A}_2,\ \mathsf{cre}(\texttt{Dir}(\texttt{nextID}^2(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{\})),\\ & \mathcal{A}_3,\ \mathsf{cre}(\texttt{Dir}(\texttt{nextID}^3(\texttt{initID}), \text{“}\texttt{source}\text{“}, \{x_1, x_2\}, \{x_0\}_s)),\\ & \mathcal{A}_4,\ \mathsf{del}(\texttt{Dir}(\texttt{nextID}^3(\texttt{initID}), \text{“}\texttt{source}\text{“}, \{x_1, x_2\}, \{x_0\}_s)),\\ & \mathcal{A}_5\rangle,\end{aligned}$$

where we have abbreviated

$$\begin{aligned} x_0 &:= \texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, ...)\\ x_1 &:= \texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\})\\ x_2 &:= \texttt{Dir}(\texttt{nextID}^2(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{\}).\end{aligned}$$

The value of `existDObj` in state $\mathcal{A}_i$ exactly corresponds to the one shown in the row corresponding to state i in Tab. 3.1.

Notice that the failed try of deleting the directory "overview" does not occur in Δ. Requirement (1) of Defn. 3.3.7(del()) does not hold for $\mathsf{del}(x_2)$ since "overview" (x_2) belongs to the object-valued content of "source".

B.2 Contexts and Concepts

Here we exemplify the formal definitions for concept syntax and semantics (Chap. 4).

Concept definitions

The formal concept definition for EntryPoint (cf. Fig. 4.1, page 55) is given as follows:

$$\begin{aligned} \mathit{ASpec} := \mathit{AWeb} &= \mathit{validAWeb}(\mathit{website}) \wedge\ \mathit{sourcedir} = \texttt{srcDir}(\mathit{website}) \wedge\\ &\quad\ \mathit{homepage} = \texttt{home}(\mathit{website})\\ \mathit{BSpec} := \mathit{BWeb} &= \mathit{validBWeb}(\mathit{website}) \wedge\ \mathit{sourcedir} = \texttt{srcDir}(\mathit{website}) \wedge\\ &\quad\ \mathit{homepage} = \texttt{home}(\mathit{website})\\ \mathcal{I} &:= (\textbf{website} : \alpha_w, \textbf{sourcedir} : \alpha_d, \textbf{homepage} : \alpha_h)\\ &\quad\ [\alpha_w{<}\texttt{Website} \wedge \alpha_d{<}\texttt{Dir} \wedge \alpha_h{<}\texttt{HTMLDoc}]\\ \mathcal{K}^{EP} &:= \mathsf{EntryPoint}\ \mathcal{I}\ =\ \{\mathit{ASpec}, \mathit{BSpec}\}\end{aligned}$$

There, *AWeb* and *BWeb* implement $\mathcal{I}$. The role assignment *ra* is given by

$$\begin{array}{ll} ra(website : \texttt{Website}) & = \textbf{website} : \alpha_w, \\ ra(sourcedir : \texttt{Dir}) & = \textbf{sourcedir} : \alpha_d, \\ ra(homepage : \texttt{HTMLDoc}) & = \textbf{homepage} : \alpha_h. \end{array}$$

Given a type binding θ, θ' yields

$$\theta' = \theta[\{\alpha_w \mapsto \texttt{Website}, \alpha_d \mapsto \texttt{Dir}, \alpha_h \mapsto \texttt{HTMLDoc}\}]$$

such that $\phi[\theta']$ becomes

$$\texttt{Website}{<}\texttt{Website} \wedge \texttt{Dir}{<}\texttt{Dir} \wedge \texttt{HTMLDoc}{<}\texttt{HTMLDoc}$$

which is obviously satisfiable.

The functional concept Name (lower left-hand part of Fig. 4.1) is given as follows:

$$\mathcal{K}^{Name} := \\ \mathsf{Name}(\textbf{entity} : \alpha_e) \rightarrow \texttt{String}[\alpha_e{<}\texttt{DObj}] = \{WebN = \texttt{name}(entity : \texttt{Website})\}$$

WebN implements the interface, where the role assignment *ra* yields $ra(entity : \texttt{Website}) = \textbf{entity} : \alpha_e$. There, we refer to a general named *entity* using the role $\textbf{entity} : \alpha_\textbf{e}$. The type variable α_e may be instantiated by an arbitrary object type. Yet we require the result type String. The context *WebN* instantiates α_e with Website. The concept definition for Name particularly shows similarities to function definitions in programming languages.

Dynamic Term Values

We encourage the reader to verify the results shown in Tab. 4.1, page 57, using the formal semantics provided in Tab. 4.4, page 61. Apart from that we mention that the dynamic term semantics is *tolerant* in some sense. The term $\texttt{name}(\texttt{Dir}(id, n, sdirs, sdocs))$, e.g, evaluates to n even if the directory does not exist. In our view, the term has no object-valued content since it can simply be represented by a constructor of type String.

As another example, take sequences that are restricted to a maximum length of one. Then the term $\langle t_1, t_2 \rangle$ would evaluate to $\langle t_1 \rangle$ if t_1 exists. Both terms represent the same sequence (a try to add t_2 fails since the the sequence is limited to length one), but the latter is shorter; we merely consider constructor terms of minimal length.

Dynamic Formula Semantics

We derive the quantifier sphere for

$$\forall w : \texttt{Website} \bullet \mathsf{Name}(w)[WebN] = \text{“}\texttt{Calculation}\text{”}.$$

in order to show how the domain mapping *dom* of Defn. 4.3.2 works. According to Defn. 4.3.2,

$$dom(\mathcal{A}, x : \tau, \eta) = (\mathcal{A}, \{v \mid v \in \tau^{\mathcal{A}}, ct \in CT^{min}_{\mathcal{A}}(\Sigma, v), (\mathcal{A}, ct) \overset{t_d}{\rightsquigarrow} v\}).$$

Applied to the formula above, this yields

$$dom(\mathcal{A}, w : \texttt{Website}, \eta) = (\mathcal{A}, \{v \mid \begin{array}{l} v \in \texttt{Website}^{\mathcal{A}}, ct \in CT^{min}_{\mathcal{A}}(\Sigma, v), \\ (\mathcal{A}, ct) \overset{t_d}{\rightsquigarrow} v\}), \end{array}$$

which is equivalent to

$$dom(\mathcal{A}, w : \texttt{Website}, \eta) = (\mathcal{A}, \begin{array}{l} \texttt{Website}^{\mathcal{A}} \cap \\ \{\mathcal{V}^{\mathcal{A}}_d[\![t]\!] \mid t \in GT_{\texttt{Website}}(\Sigma), \mathcal{A} \models t \in \texttt{existDObj}\}) \end{array}$$

according to Lemma 4.3.2(2) above. Hence, the quantifier sphere reduces to the set of existing objects of type `Website`. If "Calculation" is the only existing website, the formula above holds.

Semantics of Concept Expressions

Given a term $t_w \in T_{\texttt{Website}}(X, \Sigma)$, a state $\mathcal{A}$, and a variable assignment η, the *concept term*

$$\mathsf{Name}(t_w)[WebN]$$

is evaluated as follows according to Tab. 4.6, page 68: Using the concept definition

$$\mathsf{Name}(\textbf{website} : \alpha_e) \rightarrow \texttt{String}[\alpha_e{<}\texttt{DObj}] = \{WebN = \texttt{name}(entity)\},$$

the components of rule **Concept terms with context** in Tab. 4.6 are determined by:

$WebN = \texttt{name}(entity)$	(context definition)
$\mathcal{FV}(\texttt{name}(entity)) = \{entity : \texttt{Website}\}$	(free variables of ι_{WebN})
$l(entity : \texttt{Website}) = 1$	(index mapping w.r.t. role assignment)

Hence, the update for $entity : \texttt{Website}$ is given by $entity : \texttt{Website} \mapsto \mathcal{V}^{\mathcal{A}}_d[\![t_w]\!]\eta$ such that

$$\mathcal{V}^{\mathcal{A}}_d[\![\mathsf{Name}(t_w)[WebN]]\!]\eta = \mathcal{V}^{\mathcal{A}}_d[\![\texttt{name}(entity)]\!]\eta[entity : \texttt{Website} \mapsto \mathcal{V}^{\mathcal{A}}_d[\![t_w]\!]\eta].$$

This yields the name of the website t_w as desired.

Given terms $t_w \in T_{\texttt{Website}}(X, \Sigma), t_d \in T_{\texttt{Dir}}(X, \Sigma)$, and $t_h \in T_{\texttt{HTMLDoc}}(X, \Sigma)$, a state $\mathcal{A}$, and a variable assignment η, the *concept formula*

$$\mathsf{EntryPoint}(t_w, t_d, t_h)[AWeb]$$

is evaluated as follows: Using the parts

$$\begin{array}{rcl} AWeb = & validAWeb(website) \wedge \ sourcedir = \texttt{srcDir}(website) \wedge \\ & homepage = \texttt{home}(website) \\ \mathcal{I} & := (\textbf{website} : \alpha_w, \textbf{sourcedir} : \alpha_d, \textbf{homepage} : \alpha_h) \\ & [\alpha_w{<}\texttt{Website} \wedge \alpha_d{<}\texttt{Dir} \wedge \alpha_h{<}\texttt{HTMLDoc}] \end{array}$$

of the concept definition and the role assignment $ra(AWeb, \mathcal{I})$

$$\begin{array}{lcl} ra(website : \texttt{Website}) & = & \textbf{website} : \alpha_w, \\ ra(sourcedir : \texttt{Dir}) & = & \textbf{sourcedir} : \alpha_d, \\ ra(homepage : \texttt{HTMLDoc}) & = & \textbf{homepage} : \alpha_h, \end{array}$$

the components of **Concept formulas** in Tab. 4.6 are determined by:

$$\begin{array}{lll} AWeb & = validAWeb(website) \wedge & \text{(context definition)} \\ & \quad sourcedir = \texttt{srcDir}(website) \wedge & \\ & \quad homepage = \texttt{home}(website) & \\ \mathcal{FV}(\texttt{name}(website)) & = \{website : \texttt{Website}, sourcedir : \texttt{Dir}, & \text{(free variables of } \iota_{AWeb}) \\ & \quad homepage : \texttt{HTMLDoc}\} & \\ l(website : \texttt{Website}) = 1 & & \text{(index mapping w.r.t.} \\ l(sourcedir : \texttt{Dir}) = 2 & & \text{role assignment)} \\ l(homepage : \texttt{HTMLDoc}) = 3 & & \end{array}$$

Hence, the updates for $website, sourcedir$, and $homepage$ are given by

$$\begin{array}{lll} upd_1 := website : \texttt{Website} & \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_w]\!]\eta, \\ upd_2 := sourcedir : \texttt{Dir} & \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_d]\!]\eta, \text{ and} \\ upd_3 := homepage : \texttt{HTMLDoc} & \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_h]\!]\eta. \end{array}$$

If additionally

$$\begin{array}{ll} \mathcal{A} \models_d & validAWeb(website) \wedge\ sourcedir = \texttt{srcDir}(website) \wedge \\ & homepage = \texttt{home}(website)[\eta[upd_1][upd_2][upd_3]] \end{array}$$

we conclude $\mathcal{A} \models_d \mathsf{EntryPoint}(t_w, t_d, t_h)[AWeb][\eta]$. The variant $\mathsf{EntryPoint}(t_w, t_d, t_h)$ holds iff the website t_w has an arbitrary website format that is implemented in EntryPoint. Suppose we have specified format preservation using a wildcard. Whenever a new website format is added, this specification need not be adapted. The concept semantics automatically quantifies over all known implementations.

B.3 Formal Preservation Requirements

Here we provide examples related to the preservation predicate *pres*, traces, trace formulas, and preservation formulas as defined in Chap. 5.

Preservation Predicate

We provide examples for non-functional and functional concepts.
Preservation of non-functional concepts: In the example of the introduction to Chap. 5, o and d are transformed to o_1', o_2' and d', respectively. Preservation of the concept Contains w.r.t. o, d and o_1', d' is expressed by

$$pres(\mathsf{Contains}(o, d)[CDoc], \mathsf{Contains}(o_1', d')[CDoc])$$

According to Defn. 5.2.1, page 76, this holds if

$$\mathcal{A}_s \models_d \mathsf{Contains}(o, d)[CDoc][\eta_s] \Leftrightarrow \mathcal{A}_t \models_d \mathsf{Contains}(o_1', d')[CDoc][\eta_t].$$

Negation is interesting as it enforces a status change. If $\mathsf{Contains}(o, d)[CDoc]$ does not hold,

$$\neg pres(\mathsf{Contains}(o, d)[CDoc], \mathsf{Contains}(o_1', d')[CDoc])$$

holding enforces $\mathsf{Contains}(o'_1, d')[CDoc]$ due to the strong correspondence $\Leftrightarrow$. In particular,

$$\neg(\mathcal{A}_s \models_d \mathsf{Contains}(o, d)[CDoc][\eta_s] \Leftrightarrow \mathcal{A}_t \models_d \mathsf{Contains}(o'_1, d')[CDoc][\eta_t])$$

is equivalent to

$$\mathcal{A}_s \not\models_d \mathsf{Contains}(o, d)[CDoc][\eta_s] \Leftrightarrow \mathcal{A}_t \models_d \mathsf{Contains}(o'_1, d')[CDoc][\eta_t].$$

Preservation of functional concepts: In Chap. 4 we have introduced the specification for the concept Name. Suppose we extend it to

$$\mathsf{Name}(\mathbf{entity} : \alpha_e) \to \mathtt{String}[\alpha_e{<}\mathtt{DObj}] = \{\ \mathit{WebN} = \mathtt{name}(\mathit{entity} : \mathtt{Website})\} \\ \mathit{DirN} = \mathtt{name}(\mathit{entity} : \mathtt{Dir})\}$$

such that it additionally covers directory names in the context *DirN*. Preservation of Name for o in o'_1 can be expressed in different ways. Examples are

(1) $\mathit{pres}(\mathsf{Name}(o)[\mathit{DirN}], \mathsf{Name}(o'_1,)[\mathit{DirN}])$,
(2) $\mathit{pres}(\mathsf{Name}(o)[\_], \mathsf{Name}(o'_1,)[\mathit{DirN}])$, and
(3) $\mathit{pres}(\mathsf{Name}(o)[\_], \mathsf{Name}(o'_1,)[\_])$.

In contrast, $\mathit{pres}(\mathsf{Name}(o)[\mathit{WebN}], \mathsf{Name}(o'_1,)[\_])$ is ill-typed according to Defn. 4.3.3 as directory o cannot be matched to type `Website`.

Expressions (1) to (3) hold iff

(1) $\exists v \in \mathtt{Top}^{\mathcal{A}_s} \bullet (\mathcal{A}_s, \eta_s, \mathsf{Name}(o)[\mathit{DirN}]) \overset{t_d}{\rightsquigarrow} v \wedge (\mathcal{A}_t, \eta_t, \mathsf{Name}(o'_1)[\mathit{DirN}]) \overset{t_d}{\rightsquigarrow} v$,
(2) $\exists C \in \{\mathit{WebN}, \mathit{DirN}\}, v \in \mathtt{Top}^{\mathcal{A}_s} \bullet$
$(\mathcal{A}_s, \eta_s, \mathsf{Name}(o)[C]) \overset{t_d}{\rightsquigarrow} v \wedge (\mathcal{A}_t, \eta_t, \mathsf{Name}(o'_1)[\mathit{DirN}]) \overset{t_d}{\rightsquigarrow} v$,
(3) $\exists C, C' \in \{\mathit{WebN}, \mathit{DirN}\}, v \in \mathtt{Top}^{\mathcal{A}_s} \bullet$
$(\mathcal{A}_s, \eta_s, \mathsf{Name}(o)[C]) \overset{t_d}{\rightsquigarrow} v \wedge (\mathcal{A}_t, \eta_t, \mathsf{Name}(o'_1)[C']) \overset{t_d}{\rightsquigarrow} v$

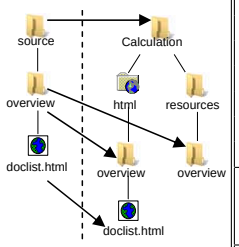
according to Defn. 5.2.1 and Defn. 4.3.4. Variant (1) reduces to

$$\exists v \in \mathtt{Top}^{\mathcal{A}_s} \bullet \\ (\mathcal{A}_s, \eta_s[\mathit{entity} : \mathtt{Dir} \mapsto \mathcal{V}_d^{\mathcal{A}_s}[\![o]\!]\eta_s], \mathtt{name}(\mathit{entity} : \mathtt{Dir})) \overset{t_d}{\rightsquigarrow} v \wedge \\ (\mathcal{A}_t, \eta_t[\mathit{entity} : \mathtt{Dir} \mapsto \mathcal{V}_d^{\mathcal{A}_t}[\![o'_1]\!]\eta_t], \mathtt{name}(\mathit{entity} : \mathtt{Dir})) \overset{t_d}{\rightsquigarrow} v.$$

As desired, the *pres* predicate holds if o and o'_1 have equal names. Since *DirN* is the only suitable context for o, o'_1, variants (2) and (3) are equivalent to variant (1). In general, however, evaluating (2) and (3) is considerably more inefficient than evaluating (1) as *all* known contexts are checked.

Negation of $\mathit{pres}(\mathsf{Name}(o)[\mathit{DirN}], \mathsf{Name}(o'_1,)[\mathit{DirN}])$ holds according to Defn. 5.2.1 if o or o'_1 do not exist or $\mathsf{Name}(o)[\mathit{DirN}]$ and $\mathsf{Name}(o'_1,)[\mathit{DirN}]$ cannot be reduced to equal values. If o and o'_1 exist, negation holds iff $\mathsf{Name}(o)[\mathit{DirN}]$ and $\mathsf{Name}(o'_1,)[\mathit{DirN}]$ are different.

Table B.1: *Example migration sequence and extracted traces*



Δ	$traces_0(\Delta)$	$step(\Delta)$	$traces_1(\Delta)$	$maxtraces(\Delta)$
$\langle \mathcal{A}_0,\ \mathsf{trans}(d \mapsto d'),$	$\langle(\mathcal{A}_0, d)\rangle$	$\langle(\mathcal{A}_0, d), (\mathcal{A}_1, d')\rangle$	$\langle(\mathcal{A}_0, d), (\mathcal{A}_1, d')\rangle$	$\langle(\mathcal{A}_0, d), (\mathcal{A}_6, d')\rangle$
$\mathcal{A}_1,\ \mathsf{trans}(o \mapsto o'_1),$	$\langle(\mathcal{A}_1, o)\rangle$	$\langle(\mathcal{A}_1, o), (\mathcal{A}_2, o'_1)\rangle$	$\langle(\mathcal{A}_1, o), (\mathcal{A}_2, o'_1)\rangle$	$\langle(\mathcal{A}_0, o), (\mathcal{A}_6, o'_1)\rangle$
$\mathcal{A}_2,\ \mathsf{trans}(o \mapsto o'_2),$		$\langle(\mathcal{A}_2, o), (\mathcal{A}_3, o'_2)\rangle$	$\langle(\mathcal{A}_2, o), (\mathcal{A}_3, o'_2)\rangle$	$\langle(\mathcal{A}_0, o), (\mathcal{A}_6, o'_2)\rangle$
$\mathcal{A}_3,\ \mathsf{cre}(h),$				
$\mathcal{A}_4,\ \mathsf{cre}(r),$				
$\mathcal{A}_5,\ \mathsf{trans}(s \mapsto c),$	$\langle(\mathcal{A}_0, s)\rangle$	$\langle(\mathcal{A}_5, s), (\mathcal{A}_6, c)\rangle$	$\langle(\mathcal{A}_5, s), (\mathcal{A}_6, c)\rangle$	$\langle(\mathcal{A}_0, s), (\mathcal{A}_6, c)\rangle$
$\mathcal{A}_6\rangle$				
$\begin{pmatrix}\langle(\mathcal{A}_0, d), (\mathcal{A}_6, d')\rangle,\\ \langle(\mathcal{A}_0, o), (\mathcal{A}_6, o'_1)\rangle,\\ \langle(\mathcal{A}_0, s), (\mathcal{A}_6, c)\rangle\end{pmatrix}$			$\begin{pmatrix}\langle(\mathcal{A}_0, d), (\mathcal{A}_6, d')\rangle,\\ \langle(\mathcal{A}_0, o), (\mathcal{A}_6, o'_2)\rangle,\\ \langle(\mathcal{A}_0, s), (\mathcal{A}_6, c)\rangle\end{pmatrix}$	
$\mathcal{TR}(\Delta)$				

Object Traces

In Tab. B.1 we show an excerpt of the example website transformation explained in the introduction to Chap. 5. The underlying transformations are visualized by arrows in the left-hand part. In particular, we transform the directory / file chain starting at "source" and ending at "doclist.html". The table in the right-hand part shows the formal transformation sequence Δ, where we have abbreviated

d, d'	–	html file "doclist.html" and its transformation result,
o, o'_1, o'_2	–	directory "overview" and its transformation results,
h	–	directory "html",
r	–	directory "resources", and
s, c	–	directory "source" and its transformation result "Calculation".

We assume that only the source objects d, o, s exist in $\mathcal{A}_0$. The set of traces of length zero, thus, contains those traces that are shown in the column labeled $traces_0(\Delta)$. The set $step(\Delta)$ represents the single transformation steps of Δ. Recall that it is used to generate traces of length $i+1$ from traces of length i. For all i, the element in row i in column $traces_1$ is determined by $tr_j \textcircled{:} tr'_i$, where tr_j and tr'_i denote the element in row j in column $traces_0(\Delta)$ and the element in row i in column $step(\Delta)$, respectively, such that the source of tr_j equals the source of tr'_i.

The set of inclusionmaximal traces $maxtraces(\Delta)$ is shown in the right-most column. In this example, it is similar to $traces_1(\Delta)$. However, all traces in $maxtraces(\Delta)$ start in state $src(\Delta)$ and end in state $res(\Delta)$. Finally, the set $\mathcal{TR}(\Delta)$ contains the trace tuples shown in the lower right-hand part of Tab. B.1. They cover inclusionmaximal histories for all existing objects in parallel.

Trace Formulas

Here we explain syntactic and semantic aspects of some selected trace formulas in more detail.

Syntax: The transformation constraint

$$(1)\quad \texttt{Dir}(\texttt{nextID}^2(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{d\}) \mapsto \texttt{Dir}$$

requires "overview" to be transformed to type `Dir` and is well-formed according to Defn. 5.4.1 and Tab. 5.1; the prerequisites $\texttt{Dir}(\texttt{nextID}^2(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{d\}) \in$

$T_{\texttt{DObj}}(X, \Sigma)$ and $\texttt{Dir}{<}\texttt{DObj}$ hold.
The functional preservation constraint $\texttt{pres}_f(\mathsf{Name}(o)[C_s, C_t])$ is well-formed, provided that $\mathsf{Name}(o)[C_s]$ is in $KT(X, \Sigma, KD)$ and $C_t \in \{\mathit{WebN}, \mathit{DirN}, \_\}$. Hence, permissible variants include

(2) $\texttt{pres}_f(\mathsf{Name}(o)[\mathit{DirN}, \mathit{DirN}])$,
(3) $\texttt{pres}_f(\mathsf{Name}(o)[\mathit{DirN}, \mathit{WebN}])$, and
(4) $\texttt{pres}_f(\mathsf{Name}(o)[\_, \_])$.

Well-formedness rules for non-functional concept preservation constraints very much correspond to those for functional concept preservation constraints. In particular,

$$\texttt{pres}_{nf}(\mathsf{Contains}(o, d)[\mathit{CDoc}, \mathit{CDoc}])$$

is well-formed since $\mathsf{Contains}(o, d)[\mathit{CDoc}]$ is a valid concept formula in $KF(X, \Sigma, KD)$ and *CDoc* is a context name of Contains.

Using conjunction, negation, and quantification, more complex trace formulas can be composed from these atomic ones. Examples are:

(5) $\neg\texttt{Dir}(\texttt{nextID}^2(\texttt{initID}), \text{“}\texttt{overview}\text{“}, \{\}, \{d\}) \mapsto \texttt{DObj}$,
(6) $\neg\texttt{pres}_{nf}(\mathsf{Contains}(o, d)[\mathit{CDoc}, \mathit{CDoc}])$, and
(7) $\forall h : \texttt{HTMLDoc} \bullet o \mapsto \texttt{Dir} \wedge \texttt{pres}_{nf}(\mathsf{Contains}(o, h)[\mathit{CDoc}, \mathit{CDoc}])$.

Formula (5) prevents “overview” from being transformed. Trace formula (6) says that the status of directory containment between o and d must not be preserved. According to formula (7) the Contains concept is to be preserved for o, all html files h and *CDoc* in context *CDoc*.

Semantics: In the following we use the trace

$$tr := \begin{pmatrix} \langle(\mathcal{A}_0, d), (\mathcal{A}_6, d')\rangle, \\ \langle(\mathcal{A}_0, o), (\mathcal{A}_6, o'_1)\rangle, \\ \langle(\mathcal{A}_0, s), (\mathcal{A}_6, c)\rangle \end{pmatrix}$$

of Fig. B.1, page 189 in order to determine the semantics of some of the trace formulas above. According to the rules in Tab. 5.1,

$$\frac{\begin{array}{ll} o \mapsto \texttt{Dir} \in TF^{at}(X, \Sigma, KD) & tr \in \bigcup_{\Delta \in MS(\Sigma)} \mathcal{TR}(\Delta) \\ t' \equiv \mathit{version}(|\pi_o(tr)|, \pi_o(tr)) & \eta \in Env(X, \mathit{src}(tr)) \\ |\pi_o(tr)| \geq 1 & \mathit{res}(tr) \models_d \texttt{inst}_{\texttt{Dir}}(t')[\eta] \end{array}}{tr \models_d o \mapsto \texttt{Dir}[\eta]}$$

Since

$$\begin{array}{lll} \pi_o(tr) & = & \langle(\mathcal{A}_0, o), (\mathcal{A}_6, o'_1)\rangle, \\ \mathit{version}(|\pi_o(tr)|, \pi_o(tr)) & = & o'_1 \\ \mathit{src}(tr) & = & \mathcal{A}_0 \\ \mathit{res}(tr) & = & \mathcal{A}_6 \end{array}$$

this rule is equivalent to

$$\frac{\begin{array}{ll} o \mapsto \texttt{Dir} \in TF^{at}(X, \Sigma, KD) & \eta \in Env(X, \mathcal{A}_0) \\ |\langle(\mathcal{A}_0, o), (\mathcal{A}_6, o'_1)\rangle| \geq 1 & \mathcal{A}_6 \models_d \texttt{inst}_{\texttt{Dir}}(o'_1)[\eta] \end{array}}{tr \models_d o \mapsto \texttt{Dir}[\eta].}$$

Hence, this transformation constraint is satisfied for tr and has the desired semantics: The last version o'_1 of the trace for o has type `Dir` and o is indeed transformed; its trace has a length greater or equal to one.

Constraint (5) above holds if $tr \models_d o \mapsto \texttt{DObj}[\eta]$ does not hold, i.e. the rule

$$\frac{\begin{array}{ll} o \mapsto \texttt{DObj} \in TF^{at}(X, \Sigma, KD) & \eta \in Env(X, \mathcal{A}_0) \\ |\pi_o(tr)| \geq 1 & \mathcal{A}_6 \models_d \texttt{inst}_{\texttt{DObj}}(o'_1)[\eta] \end{array}}{tr \models_d o \mapsto \texttt{Dir}[\eta].}$$

must not be applicable. This implies $|\pi_o(tr)| \not\geq 1$ (i.e., o is not transformed) since all other prerequisites of the rule hold. In particular, $\mathcal{A} \models_d \texttt{inst}_{\texttt{DObj}}(x)[\eta]$ is a tautology for all existing objects.

Constraint (3) ($\texttt{pres}_f(\mathsf{Name}(o)[DirN, WebN])$) is evaluated using the rule

$$\frac{\begin{array}{ll} \texttt{pres}_f(\mathsf{Name}(o)[DirN, WebN]) \in TF^{at}(X, \Sigma, KD) & tr \in \bigcup_{\Delta \in MS(\Sigma)} \mathcal{TR}(\Delta) \\ t'_1 \equiv version(|\pi_o(tr)|, \pi_o(tr)) & \mathcal{A} = src(\pi_o(tr)),\ \mathcal{A}' = res(\pi_o(tr)) \\ & \eta \in Env(X, \mathcal{A}) \\ & (\mathcal{A}, \mathcal{A}') \models_d pres(\mathsf{Name}(o)[DirN], \mathsf{Name}(t'_1)[WebN])[\eta, \eta] \end{array}}{tr \models_d \texttt{pres}_f(\mathsf{Name}(o)[DirN, WebN])[\eta].}$$

Since

$$\begin{array}{ll} \pi_o(tr) & = \langle (\mathcal{A}_0, o), (\mathcal{A}_6, o'_1) \rangle, \\ version(|\pi_o(tr)|, \pi_o(tr)) & = o'_1, \\ src(\pi_o(tr)) & = \mathcal{A}_0, \\ res(\pi_o(tr)) & = \mathcal{A}_6, \end{array}$$

this rule is equivalent to

$$\frac{\begin{array}{ll} \texttt{pres}_f(\mathsf{Name}(o)[DirN, WebN]) \in TF^{at}(X, \Sigma, KD) & \eta \in Env(X, \mathcal{A}_0) \\ & (\mathcal{A}_0, \mathcal{A}_6) \models_d pres(\mathsf{Name}(o)[DirN], \mathsf{Name}(o'_1)[WebN])[\eta, \eta] \end{array}}{tr \models_d \texttt{pres}_f(\mathsf{Name}(o)[DirN, WebN])[\eta].}$$

This constraint does *not* hold. The trace result o'_1 is not of type `Website`. Hence, it cannot be matched to Name in *WebN*. In general, this constraint does not hold before o is transformed to type `Website`; preservation constraints can enforce transformations. Notice, however, that preservation constraints cannot *forbid* transformations. Hence, transformation constraints are not purely syntactic sugar.

The other two variants

$$\texttt{pres}_f(\mathsf{Name}(o)[DirN, DirN]) \text{ and } \texttt{pres}_f(\mathsf{Name}(o)[\_, \_])$$

from above hold. This results from substituting the contexts appropriately in the just-provided rule.

Recall that negated preservation constraints enforce a status change. In the following we derive the semantics for constraint (6) from above in order to have an example. Applying rule **Negation** (Tab. 3.3, page 40) we get

$$\frac{\begin{array}{ll} \texttt{pres}_{nf}(\mathsf{Contains}(o, d)[CDoc, CDoc]) \in TF(X, \Sigma, KD) & \text{not } tr \models_d \texttt{pres}_{nf}(\mathsf{Contains}(o, d)[CDoc, CDoc])[\eta] \end{array}}{tr \models_d \neg\texttt{pres}_{nf}(\mathsf{Contains}(o, d)[CDoc, CDoc])[\eta].}$$

Applying rule **Non-functional concept preservation constraints** we get

$$\frac{\begin{array}{l} \texttt{pres}_{nf}(\mathsf{Contains}(o, d)[CDoc, CDoc]) \in TF(X, \Sigma, KD) \\ (\mathcal{A}_0, \mathcal{A}_6) \models_d pres(\ \mathsf{Contains}(o, d)[CDoc], \\ \qquad\qquad \mathsf{Contains}(o'_1, d')[CDoc])[\eta, \eta] \end{array}}{tr \models_d \texttt{pres}_{nf}(\mathsf{Contains}(o, d)[CDoc, CDoc])[\eta].}$$

Since $\mathtt{pres}_{nf}(\mathsf{Contains}(o,d)[CDoc,CDoc]) \in TF(X,\Sigma,KD)$ is true,

$$tr \models_d \mathtt{pres}_{nf}(\mathsf{Contains}(o,d)[CDoc,CDoc])[\eta]$$

holds unless $(\mathcal{A}_0,\mathcal{A}_6) \not\models_d pres(\mathsf{Contains}(o,d)[CDoc],\mathsf{Contains}(o'_1,d')[CDoc])[\eta,\eta]$. According to Defn. 5.2.1, this is equivalent to

$$\mathcal{A}_0 \not\models_d \mathsf{Contains}(o,d)[CDoc][\eta] \Leftrightarrow \mathcal{A}_6 \models_d \mathsf{Contains}(o'_1,d')[CDoc][\eta],$$

since all affected objects exist in the system. This corresponds to the just-mentioned status change.

We conclude this example by deriving the semantics of trace formula (7) from above. Applying rule **Quantification** (Tab. 3.3, page 40) we get

$$\frac{\begin{array}{ll}\forall h : \mathtt{HTMLDoc} \bullet & dom(tr,h,\eta) = (\mathcal{A},d)\\ \quad o \mapsto \mathtt{Dir} \wedge & tr \models_d o \mapsto \mathtt{Dir} \wedge\\ \quad \mathtt{pres}_{nf}(\mathsf{Contains}(o,h)[CDoc,CDoc]) & \quad \mathtt{pres}_{nf}(\mathsf{Contains}(o,h)[CDoc,CDoc])\\ \in TF(X,\Sigma,KD) & \quad [\eta[h \mapsto a]]\\ & \text{for all } a \in d\end{array}}{tr \models_d \forall h : \mathtt{HTMLDoc} \bullet o \mapsto \mathtt{Dir} \wedge \mathtt{pres}_{nf}(\mathsf{Contains}(o,h)[CDoc,CDoc])[\eta].}$$

In Defn. 5.4.1 *dom* is defined by

$$dom(tr,x:\tau,\eta) := (src(tr),\{v \mid\ v \in \tau^{src(tr)}, ct \in CT^{min}_{src(tr)}(\Sigma,v), (src(tr),ct) \overset{t_d}{\rightsquigarrow} v\}).$$

Applied to this example,

$$dom(tr,h:\mathtt{HTMLDoc},\eta) = (\mathcal{A}_0,\{v \mid\ v \in \mathtt{HTMLDoc}^{\mathcal{A}_0}, ct \in CT^{min}_{\mathcal{A}_0}(\Sigma,v), (\mathcal{A}_0,ct) \overset{t_d}{\rightsquigarrow} v\}).$$

Assuming d is the only html file that exists in $\mathcal{A}_0$, $dom(tr,h:\mathtt{HTMLDoc},\eta)$ evaluates to $(\mathcal{A}_0,\{\mathcal{V}_d^{\mathcal{A}_0}[\![d]\!]\eta\}$. Substituting this in the rule above results in

$$\frac{\begin{array}{ll}\forall h : \mathtt{HTMLDoc} \bullet & tr \models_d o \mapsto \mathtt{Dir} \wedge\\ \quad o \mapsto \mathtt{Dir} \wedge & \quad \mathtt{pres}_{nf}(\mathsf{Contains}(o,h)[CDoc,CDoc])\\ \quad \mathtt{pres}_{nf}(\mathsf{Contains}(o,h)[CDoc,CDoc]) & \quad [\eta[h \mapsto a]]\\ \in TF(X,\Sigma,KD) & \text{for all } a \in \{\mathcal{V}_d^{\mathcal{A}_0}[\![d]\!]\eta\}\end{array}}{tr \models_d \forall h : \mathtt{HTMLDoc} \bullet o \mapsto \mathtt{Dir} \wedge \mathtt{pres}_{nf}(\mathsf{Contains}(o,h)[CDoc,CDoc])[\eta]}$$

This is the desired semantics — $\mathsf{Contains}(o,h)$ is to be preserved for all *existing* html files h while o is transformed to type `Dir`. Since traces contain *one explicit* history chain for the objects that exist in the source state,

$$tr \models_d \forall h : \mathtt{HTMLDoc} \bullet o \mapsto \mathtt{Dir} \wedge \mathtt{pres}_{nf}(\mathsf{Contains}(o,h)[CDoc,CDoc])[\eta]$$

holding assures that *one* transformation result of o satisfies the constraint

$$\mathtt{pres}_{nf}(\mathsf{Contains}(o,h)[CDoc,CDoc])$$

for one transformation result of *each* existing html file.

Preservation Formulas

In the following we list some selected examples for preservation formulas. As their evaluation reduces to evaluating Σ-terms, concept formulas and trace formulas in a straightforward way (cf. Tab. 5.2 on page 83), we do not derive their semantics explicitly.

Apart from those examples that have already been provided previously, the following are well-formed preservation formulas:

(1) $\forall d : \texttt{Dir} \bullet \textcircled{\forall} \neg d \mapsto \texttt{DObj} \Rightarrow \texttt{pres}_f(\mathit{Name}(d)[\_, \_])$
(2) $\forall d : \texttt{Dir} \bullet \textcircled{\exists} d \mapsto \texttt{Dir} \wedge \forall h : \texttt{HTMLDoc} \bullet \texttt{pres}_{nf}(\mathsf{Contains}(d, h)[\mathit{CDoc}, \mathit{CDoc}])$
(3) $\forall w : \texttt{Website} \bullet$
$\quad \forall d, d' : \texttt{Dir} \bullet$
$\quad\quad d = \texttt{srcDir}(w) \Rightarrow$
$\quad\quad\quad \textcircled{\exists} \; d \mapsto \texttt{Dir} \wedge d' \mapsto \texttt{Dir} \wedge \texttt{pres}_{nf}(\mathsf{Contains}(d, d')[\mathit{CDirRec}, \mathit{CDirHtml}])$

There, formula (1) states that directory names are to be preserved for those directories that are not transformed. This is a tautology. Formula (1) even scales to all object types and all concepts the value of which depend on the arguments only. This is often referred to as *referential transparency* in the literature. Generally speaking, formula (1) reflects object immutability in our system; an object's properties cannot change unless it is transformed.

Preservation formula (2) extends formula (7) of the previous example for trace formulas. It quantifies over all directories and requires a trace to *exist* for each directory d such that

(1) d is transformed and
(2) the concept Contains is preserved for all html files and d w.r.t. that trace.

This shows the difference between quantifying outside and inside of trace formulas, respectively. There may be different traces for different directories d. The html files, however, are determined w.r.t. the same trace.

Finally, formula (3) is a modification of the above shown preservation requirement of the introduction to Chap. 5. Instead of $\mathsf{WebSrc}(d, w)$ we have used $d = \texttt{srcDir}(w)$; we permit regular Σ-terms in preservation formulas. Yet notice that formula (3) reveals implementation details of WebSrc, which can make the preservation requirement prone to changing implementations of WebSrc.

B.4 Migration Algorithms

Here we explain syntactic and semantic aspects of basic operations and migration algorithms in more detail.

Basic Operations

The sample archive evolution in Tab. 6.1 contains a series of basic operations. When fully expanded and executed in a state $\mathcal{A}$, these operations look as follows:

(1) `create`(`HTMLDoc`, “`start.html`“, ..., $\emptyset$)
(2) `create`(`Dir`, “`calc05`“, {}, {})
(3) `create`(`Dir`, “`overview`“, {}, {})
(4) `create`(`Dir`, “`source`“, $\{x_1, x_2\}$, $\{x_0\}_s$)
$\eta(x_0) = \mathcal{V}^{\mathcal{A}}[\![$`HTMLDoc`(`initID`, “`start.html`“, ...)$]\!]$
$\eta(x_1) = \mathcal{V}^{\mathcal{A}}[\![$`Dir`(`nextID`(`initID`), “`calc05`“, {}, {})$]\!]$
$\eta(x_2) = \mathcal{V}^{\mathcal{A}}[\![$`Dir`((`nextID`^{2}(`initID`), “`overview`“, {}, {})$]\!]$
(5) `delete`(`Dir`((`nextID`^{2}(`initID`), “`overview`“, {}, {}))
(6) `delete`(`Dir`, “`source`“),
$\eta(x_0) = \mathcal{V}^{\mathcal{A}}[\![$`HTMLDoc`(`initID`, “`start.html`“, ...)$]\!]$
$\eta(x_1) = \mathcal{V}^{\mathcal{A}}[\![$`Dir`(`nextID`(`initID`), “`calc05`“, {}, {})$]\!]$
$\eta(x_2) = \mathcal{V}^{\mathcal{A}}[\![$`Dir`((`nextID`^{2}(`initID`), “`overview`“, {}, {})$]\!]$

The operation

`transform`(`Dir`(`nextID`(`initID`), “`calc05`“, {}, {}) $\mapsto$ (`Dir`, “`calc06`“, {}, {}))

transforms the directory “calc05” to a new directory “calc06”.

The operation `create`(`Dir`, “`test`“, {}) is ill-formed because it has too few parameters for the constructor `Dir`. As another negative example, `create`(`Dir`, “`test`“, {}, `Empty`) is ill-formed. It has the right number of parameters but `Empty` is not of type `Set`.

Effect of Basic ASM Transition Rules

The example archive evolution of Tab. 6.1 on page 89 is the result of applying the formal ASM rules

(1) *r_create*(`HTMLDoc`, “`start.html`“, ..., $\emptyset$)
(2) *r_create*(`Dir`, “`calc05`“, {}, {}, $\emptyset$)
(3) *r_create*(`Dir`, “`overview`“, {}, {}, $\emptyset$)
(4) *r_create*(`Dir`, “`source`“,
{`Dir`(`nextID`(`initID`), “`calc05`“, {}, {}),
`Dir`((`nextID`^{2}(`initID`), “`overview`“, {}, {})},
{`HTMLDoc`(`initID`, “`start.html`“, ...)}$_s$,
{`Dir`(`nextID`(`initID`), “`calc05`“, {}, {}),
`Dir`((`nextID`^{2}(`initID`), “`overview`“, {}, {}),
`HTMLDoc`(`initID`, “`start.html`“, ...)})
(5) *r_delete*(`Dir`((`nextID`^{2}(`initID`), “`overview`“, {}, {}))
(6) *r_delete*(`Dir`, “`source`“,
{`Dir`(`nextID`(`initID`), “`calc05`“, {}, {}),
`Dir`((`nextID`^{2}(`initID`), “`overview`“, {}, {})},
{`HTMLDoc`(`initID`, “`start.html`“, ...)}$_s$)

consecutively. For clarity, we have distinguished the empty set constructor {} of type `Set` and the empty set $\emptyset$ in the example above. The former denotes a parameter in a constructor call (e.g., `Dir`(`nextID`(`initID`), “`calc05`“, {}, {})). The latter denotes an empty set t_{sub} of subterms of type `DObj` as it is used in the rule declarations above.

Effect of Basic Operations

We recapitulate an excerpt of the example archive evolution of Tab. 6.1 on page 89 in order to demonstrate the effect of the $\overset{op}{\rightsquigarrow}$-relation. The basic operations (1), (5), and (6) have the following semantics:

$$
\begin{array}{ll}
(1) & [\![r\_create(\texttt{HTMLDoc}, \text{“}\texttt{start.html}\text{“}, ..., \emptyset)]\!]^{\mathcal{A}}_{\eta} = U \neq \emptyset, \\
 & (\mathcal{A}, \eta, \texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, ...)) \overset{t}{\rightsquigarrow} v \\
\longrightarrow & (\mathcal{A}, \Delta, \eta, \texttt{create}(\texttt{HTMLDoc}, \text{“}\texttt{start.html}\text{“}, ...)) \overset{op}{\rightsquigarrow} \\
 & (\mathcal{A} + U, \Delta; \langle \mathcal{A}, \texttt{cre}(\texttt{HTMLDoc}(\texttt{initID}, \text{“}\texttt{start.html}\text{“}, ...)), \mathcal{A} + U \rangle, v) \\
(5) & [\![r\_delete(\texttt{Dir}, \text{“}\texttt{overview}\text{“}, \{\}, \{\})]\!]^{\mathcal{A}}_{\eta} = U = \emptyset, \\
\longrightarrow & (\mathcal{A}, \Delta, \eta, \texttt{delete}(\texttt{Dir}, \text{“}\texttt{overview}\text{“}, \{\}, \{\})) \overset{op}{\rightsquigarrow} (\mathcal{A}, \Delta, \texttt{False}^{\mathcal{A}}) \\
(6) & [\![r\_delete(\texttt{Dir}, \text{“}\texttt{source}\text{“}, \{x_1, x_2\}, \{x_0\}_s)]\!]^{\mathcal{A}}_{\eta} = U \neq \emptyset, \\
\longrightarrow & (\mathcal{A}, \Delta, \eta, \texttt{delete}(\texttt{Dir}, \text{“}\texttt{source}\text{“}, \{x_1, x_2\}, \{x_0\}_s)) \overset{op}{\rightsquigarrow} \\
 & (\mathcal{A} + U, \Delta; \langle \mathcal{A}, \texttt{del}(\texttt{Dir}(\texttt{nextID}^3(\texttt{initID}), \text{“}\texttt{source}\text{“}, \{x_1, x_2\}, \{x_0\}_s)), \mathcal{A} + U \rangle, \texttt{True}^{\mathcal{A}})
\end{array}
$$

Only the most relevant parts of the rules in Tab. 6.5 are shown. The $\longrightarrow$ arrow indicates derivation. Notice particularly operation (5). The migration sequence Δ stays unchanged and `False` is returned in order to report failure. Since the “overview” directory is in the object-valued content of the “source” directory, the pre-condition of the rule *r_delete* is not satisfied. Firing rule *r_delete*, thus, yields `skip`, which leads to an empty update set U.

Source of Ambiguity in Set Comprehension Semantics

Consider the following constellation:
$e := \{x \mid x \leftarrow d',\ d = d'\}$
$\eta = \{d \mapsto \mathcal{V}^{\mathcal{A}}[\![\{\texttt{Dir}(\texttt{initID}, \text{“}\texttt{d1}\text{“}, \{\}, \{\})\}_s]\!],$
$\quad d' \mapsto \mathcal{V}^{\mathcal{A}}[\![\{\texttt{Dir}(\texttt{initID}, \text{“}\texttt{d2}\text{“}, \{\}, \{\})\}_s]\!]\}.$
There, d and d' yield singleton sets of differently named directories. According to Tab. 6.7, evaluating $(\mathcal{A}, \eta, e)$ corresponds to evaluating a let construct of the form `let` $y = d'$ `in`.... Now suppose $y \equiv d$. This is indeed possible in our example since y and d are of type `Set`[`Dir`]. Then y is evaluated to $\eta(d')$ in η, i.e., to $\mathcal{V}^{\mathcal{A}}[\![\{\texttt{Dir}(\texttt{initID}, \text{“}\texttt{d2}\text{“}, \{\}, \{\})\}_s]\!]$. The `let` semantics now evaluates the `if`-expression using $\eta' := \eta[d \mapsto \eta(d')]$. This results in
$\eta' = \{d \mapsto \mathcal{V}^{\mathcal{A}}[\![\{\texttt{Dir}(\texttt{initID}, \text{“}\texttt{d2}\text{“}, \{\}, \{\})\}_s]\!],$
$\quad d' \mapsto \mathcal{V}^{\mathcal{A}}[\![\{\texttt{Dir}(\texttt{initID}, \text{“}\texttt{d2}\text{“}, \{\}, \{\})\}_s]\!]\}.$
Hence, the expression $e_B := d = d'$ has semantics `True`, which has not been intended originally. With a distinct variable y, the `if`-expression would have been evaluated with
$\eta = \{d \mapsto \mathcal{V}^{\mathcal{A}}[\![\{\texttt{Dir}(\texttt{initID}, \text{“}\texttt{d1}\text{“}, \{\}, \{\})\}_s]\!],$
$\quad d' \mapsto \mathcal{V}^{\mathcal{A}}[\![\{\texttt{Dir}(\texttt{initID}, \text{“}\texttt{d2}\text{“}, \{\}, \{\})\}_s]\!],$
$\quad y \mapsto \mathcal{V}^{\mathcal{A}}[\![\{\texttt{Dir}(\texttt{initID}, \text{“}\texttt{d2}\text{“}, \{\}, \{\})\}_s]\!]\}$
such that $d = d'$ is `False`. This is the intended semantics of e.

Sample Migration Algorithm

The algorithm covering the functions in Fig. 6.1, page 90, is described by $A_{ex} := (FD_{ex}, f_{main}) \in \mathcal{A}lg(X, \Sigma, \mathcal{F}_{decl})$, where

$$
\begin{array}{ll}
\mathcal{F}_{decl} := \{ & \texttt{migDirHTML} \ : \ \texttt{Dir} \to \texttt{Website} \to \texttt{Dir} \\
& \texttt{migHTMLDoc} \ : \ \texttt{HTMLDoc} \to \texttt{Website} \to \texttt{HTMLDoc} \ \}, \\
FD_{ex} := \{ & \\
& \texttt{migDirHTML}(d, w) \ = \\
& \quad \texttt{let} \ sDirs = \{\texttt{migDirHTML}(d', w) \mid d' \leftarrow \texttt{subDirs}(d), \ \texttt{True}\} \\
& \qquad sDocs = \{\texttt{migHTMLDoc}(d', w) \mid d' \leftarrow \texttt{subDocs}(d), \\
& \qquad\qquad\qquad\qquad and(not(d' = \texttt{home}(w)), \texttt{inst}_{\texttt{HTMLDoc}}(d'))\} \\
& \quad \texttt{in} \ \ \texttt{transform}(d \mapsto (\texttt{Dir}, \texttt{name}(d), sDirs, sDocs)), \\
& \texttt{migHTMLDoc}(d, w) \ = \\
& \quad \texttt{if} \ not(d = \texttt{home}(w)) \\
& \qquad \texttt{then} \ \texttt{transform}(d \mapsto (\texttt{HTMLDoc}, \texttt{name}(d), \texttt{content}(d))) \\
& \qquad \texttt{else} \ \texttt{transform}(d \mapsto (\texttt{HTMLDoc}, \text{“}\texttt{index.html}\text{“}, \texttt{content}(d))) \quad \ \} \\
f_{main} := & \ \texttt{migDirHTML} \ : \ \texttt{Dir} \to \texttt{Website} \to \texttt{Dir}
\end{array}
$$

Deriving Well-Formedness

We derive well-formedness of the sample algorithm A_{ex} of App. B.4, page 195. This corresponds to showing $FD_{ex} \subseteq FD(X, \Sigma, \mathcal{F}_{decl})$, which will be done in two steps:

Step 1: Show $\texttt{migHTMLDoc}(d, w) = e_1 \in FD_{\texttt{HTMLDoc}}(X, \Sigma, \mathcal{F}_{decl})$.
Step 2: Show $\texttt{migDirHTML}(d, w) = e_2 \in FD_{\texttt{Dir}}(\Sigma, X, \mathcal{F}_{decl})$.

Since e_1 and e_2 contain all syntactic elements of our language, both derivations together should generate a feeling for syntactic program correctness and static typing. We derive these properties in consecutive order.

Step 1:

$$
\dfrac{\dfrac{\begin{array}{l} not(d = \texttt{home}(w)) \in E_{\texttt{Bool}}(X, \Sigma, \mathcal{F}_{decl}) \\ \texttt{transform}(d \mapsto (\texttt{HTMLDoc}, ...)) \in E_{\texttt{HTMLDoc}}(X, \Sigma, \mathcal{F}_{decl}) \\ \texttt{transform}(d \mapsto (\texttt{HTMLDoc}, ...)) \in E_{\texttt{HTMLDoc}}(X, \Sigma, \mathcal{F}_{decl}) \end{array}}{\begin{array}{l} \texttt{if} \ not(d = \texttt{home}(w)) \\ \quad \texttt{then} \ \texttt{transform}(d \mapsto (\texttt{HTMLDoc}, ...)) \\ \quad \texttt{else} \ \texttt{transform}(d \mapsto (\texttt{HTMLDoc}, ...)) \\ \in E_{\texttt{HTMLDoc}}(X, \Sigma, \mathcal{F}_{decl}) \end{array}} \qquad \begin{array}{l} \texttt{migHTMLDoc} : \texttt{HTMLDoc} \times \texttt{Website} \to \\ \qquad \texttt{HTMLDoc} \in \mathcal{F}_{decl} \end{array}}{\begin{array}{l} d : \texttt{HTMLDoc} \in X, w : \texttt{HTMLDoc} \in X \end{array} \qquad \{d, w\} \subseteq \{d, w\}}
$$

$$
\begin{array}{l}
\texttt{migHTMLDoc}(d, w) \ = \\
\texttt{if} \ not(d = \texttt{home}(w)) \\
\quad \texttt{then} \ \texttt{transform}(d \mapsto (\texttt{HTMLDoc}, \texttt{name}(d), \texttt{content}(d))) \\
\quad \texttt{else} \ \texttt{transform}(d \mapsto (\texttt{HTMLDoc}, \text{“}\texttt{index.html}\text{“}, \texttt{content}(d)) \in FD_{\texttt{HTMLDoc}}(X, \Sigma, \mathcal{F}_{decl}))
\end{array}
$$

Hence, there are three derivation steps left. We start by deriving $not(d = \texttt{home}(w)) \in E_{\texttt{Bool}}(X, \Sigma, \mathcal{F}_{decl})$ and denote $gen(\Sigma) \cup (\emptyset, \emptyset, \emptyset, \emptyset, \mathcal{F}_{decl})$ by Σ' and the components of Σ' by $\mathcal{F}', \mathcal{P}'$ etc.

$$
\dfrac{\dfrac{\dfrac{d : \texttt{HTMLDoc} \in X}{d \in T_{\texttt{HTMLDoc}}(X, \Sigma')} \ \texttt{HTMLDoc<Top}}{d \in T_{\texttt{Top}}(X, \Sigma')} \qquad \dfrac{\dfrac{\dfrac{w : \texttt{Website} \in X}{w \in T_{\texttt{Website}}(X, \Sigma')} \qquad \texttt{home}_{\texttt{Website} \to \texttt{HTMLDoc}} \in \mathcal{F}'}{\texttt{home}(w) \in T_{\texttt{HTMLDoc}}(X, \Sigma')} \ \texttt{HTMLDoc<Top}}{\texttt{home}(w) \in T_{\texttt{Top}}(X, \Sigma')}}{\dfrac{=_{\texttt{Top} \times \texttt{Top} \to \texttt{Bool}} \in \mathcal{F}'}{\begin{array}{l} d = \texttt{home}(w) \in T_{\texttt{Bool}}(X, \Sigma') \\ not_{\texttt{Bool} \to \texttt{Bool}} \in \mathcal{F}' \end{array}}}{\dfrac{not(d = \texttt{home}(w)) \in T_{\texttt{Bool}}(X, \Sigma')}{not(d = \texttt{home}(w)) \in E_{\texttt{Bool}}(X, \Sigma)}}
$$

The derivation for $\texttt{transform}(d \mapsto (\texttt{HTMLDoc}, \texttt{name}(d), \texttt{content}(d))) \in E_{\texttt{HTMLDoc}}(X, \Sigma, \mathcal{F}_{decl})$ looks as follows (we denote $gen(\Sigma)$ by Σ'):

$$\dfrac{\dfrac{\dfrac{\dots}{\texttt{name}(d) \in T_{\texttt{String}}(X,\Sigma')} \quad \dfrac{\dots}{\texttt{content}(d) \in T_{\texttt{HTMLElem}}(X,\Sigma')} \quad \texttt{HTMLDoc} : \texttt{OID} \times \texttt{String} \times \texttt{HTMLElem} \to \texttt{HTMLDoc} \in \mathcal{C}'_{\texttt{HTMLDoc}} \quad d \in T_{\texttt{DObj}}(X,\Sigma') \quad \texttt{HTMLDoc<DObj}}{\texttt{transform}(d \mapsto (\texttt{HTMLDoc}, \texttt{name}(d), \texttt{content}(d))) \in \mathcal{BO}p_{\texttt{HTMLDoc}}(X,\Sigma')}}{\texttt{transform}(d \mapsto (\texttt{HTMLDoc}, \texttt{name}(d), \texttt{content}(d))) \in E_{\texttt{HTMLDoc}}(X,\Sigma,\mathcal{F}_{decl})}$$

where the remaining derivations are

$$\dfrac{d : \texttt{HTMLDoc} \in X}{\dfrac{d \in T_{\texttt{HTMLDoc}}(X,\Sigma') \quad \texttt{HTMLDoc<DObj}}{d \in T_{\texttt{DObj}}(X,\Sigma')}} \quad , \quad \dfrac{\dfrac{d : \texttt{HTMLDoc} \in X}{d \in T_{\texttt{HTMLDoc}}(X,\Sigma')} \quad \texttt{name}_{\texttt{HTMLDoc}\to\texttt{String}} \in \mathcal{F}'}{\texttt{name}(d) \in T_{\texttt{String}}(X,\Sigma')} \quad ,$$

and

$$\dfrac{\dfrac{d : \texttt{HTMLDoc} \in X}{d \in T_{\texttt{HTMLDoc}}(X,\Sigma')} \quad \texttt{content}_{\texttt{HTMLDoc}\to\texttt{HTMLElem}} \in \mathcal{F}'}{\texttt{content}(d) \in T_{\texttt{HTMLElem}}(X,\Sigma')} \quad .$$

Showing that $\texttt{transform}(d \mapsto (\texttt{HTMLDoc}, \text{“}\texttt{index.html}\text{“}, \texttt{content}(d)))$ is in $E_{\texttt{HTMLDoc}}(X, \Sigma, \mathcal{F}_{decl})$ works similar to the just-provided derivation. It is omitted for brevity.

Step 2: We derive $\texttt{migDirHTML}(d, w) = e_2 \in FD_{\texttt{Dir}}(\Sigma, X, \mathcal{F}_{decl})$.

The `let`-rule delivers

$$\dfrac{\begin{array}{l}\dots \quad \dots \quad \dots \\ \{\texttt{migDirHTML}(d', w) \mid d' \leftarrow \dots, \dots\} \in E_{\texttt{Set[Dir]}}(X,\Sigma') \\ \{\texttt{migHTMLDoc}(d', w) \mid d' \leftarrow \dots, \dots\} \in E_{\texttt{Set[Doc]}}(X,\Sigma') \\ \texttt{transform}(d \mapsto (\texttt{Dir}, \texttt{name}(d), \dots)) \in E_{\texttt{Dir}}(X,\Sigma') \\ sDirs : \texttt{Set[Dir]} \in X, sDocs : \texttt{Set[Doc]} \in X\end{array}}{\begin{array}{ll}\texttt{let}\, sDirs = \{\texttt{migDirHTML}(d', w) \mid d' \leftarrow \dots, \dots\} & \\ \quad sDocs = \{\texttt{migHTMLDoc}(d', w) \mid d' \leftarrow \dots, \dots\} & \\ \texttt{in transform}(d \mapsto (\texttt{Dir}, \texttt{name}(d), \dots)) & \texttt{migDirHTML}_{\texttt{Dir}\times\texttt{Website}\to\texttt{Dir}} \\ \in E_{\texttt{Dir}}(X,\Sigma'), d : \texttt{Dir} \in X, w : \texttt{Website} \in X & \in \mathcal{F}_{decl},\ \{d,w\} \subseteq \{d,w\}\end{array}}$$

$$\begin{array}{l}\texttt{migDirHTML}(d, w) = \\ \texttt{let}\, sDirs = \{\texttt{migDirHTML}(d', w) \mid d' \leftarrow \texttt{subDirs}(d),\ \texttt{True}\} \\ \quad sDocs = \{\texttt{migHTMLDoc}(d', w) \mid d' \leftarrow \texttt{subDocs}(d), \\ \qquad\qquad\qquad\qquad and(not(d' = \texttt{home}(w)), \texttt{inst}_{\texttt{HTMLDoc}}(d'))\} \\ \texttt{in transform}(d \mapsto (\texttt{Dir}, \texttt{name}(d), sDirs, sDocs)) \in FD_{\texttt{Dir}}(\Sigma, X, \mathcal{F}_{decl})\end{array}$$

resulting in three remaining proof obligations — the two set comprehension expressions and the transformation. We merely derive that

$$\{\texttt{migHTMLDoc}(d', w) \mid d' \leftarrow \texttt{subDocs}(d), \quad and(not(d' = \texttt{home}(w)), \texttt{inst}_{\texttt{HTMLDoc}}(d'))\}$$

is in $E_{\texttt{Set}[Doc]}(X, \Sigma, \mathcal{F}_{decl})$; the other set comprehension expression works similar and the above derivations already include transformations similar to the one used here.

The set comprehension rule delivers

$$\frac{\begin{array}{lll}\ldots & \ldots & \ldots\\ \texttt{subDocs}(d) \in E_{\texttt{Set[Doc]}}(X,\Sigma,\mathcal{F}_{decl}) & & \\ and(not(d' = \texttt{home}(w)), \texttt{inst}_{\texttt{HTMLDoc}}(d')) \in E_{\texttt{Bool}}(X,\Sigma,\mathcal{F}_{decl}) & & \\ \texttt{migHTMLDoc}(d', w) \in E_{\texttt{Doc}}(X,\Sigma,\mathcal{F}_{decl}) & & \\ d' : \texttt{Doc} \in X & & \end{array}}{\begin{array}{l}\{\texttt{migHTMLDoc}(d', w) \mid d' \leftarrow \texttt{subDocs}(d),\\ \qquad\qquad and(not(d' = \texttt{home}(w)), \texttt{inst}_{\texttt{HTMLDoc}}(d'))\}\\ \in E_{\texttt{Set[Doc]}}(X,\Sigma,\mathcal{F}_{decl})\end{array}}$$

such that three proof obligations remain. The simple ones are

$$\frac{d : \texttt{Dir} \in X}{d \in T_{\texttt{Dir}}(X,\Sigma')}$$

$$\frac{\dfrac{\texttt{migHTMLDoc}_{\texttt{Doc}\times\texttt{Website}\to\texttt{Doc}} \in \mathcal{F}'}{\texttt{subDocs}(d) \in T_{\texttt{Set[Doc]}}(X,\Sigma')}}{\texttt{subDocs}(d) \in E_{\texttt{Set[Doc]}}(X,\Sigma,\mathcal{F}_{decl})}$$

and

$$\frac{\dfrac{\dfrac{d' : \texttt{Doc} \in X}{d' \in T_{\texttt{Doc}}(X,\Sigma')} \quad \dfrac{w : \texttt{Website} \in X}{w \in T_{\texttt{Website}}(X,\Sigma')}}{\texttt{migHTMLDoc}(d', w) \in T_{\texttt{Doc}}(X,\Sigma')}}{\texttt{migHTMLDoc}(d', w) \in E_{\texttt{Doc}}(X,\Sigma,\mathcal{F}_{decl})}$$

such that showing $and(not(d' = \texttt{home}(w)), \texttt{inst}_{\texttt{HTMLDoc}}(d')) \in E_{\texttt{Bool}}(X,\Sigma,\mathcal{F}_{decl})$ concludes the overall derivation. The derivation tree looks as follows

$$\frac{\dfrac{\dfrac{\ldots \qquad \ldots}{not(d' = \texttt{home}(w)) \in T_{\texttt{Bool}}(X,\Sigma')} \quad \dfrac{\dfrac{d' : \texttt{HTMLDoc} \in X}{d' \in T_{\texttt{HTMLDoc}}(X,\Sigma')} \quad \texttt{HTMLDoc} \in \mathcal{T}}{\texttt{inst}_{\texttt{HTMLDoc}}(d') \in T_{\texttt{Bool}}(X,\Sigma')}}{and_{\texttt{Bool}\times\texttt{Bool}\to\texttt{Bool}} \in \mathcal{F}'}}{and(not(d' = \texttt{home}(w)), \texttt{inst}_{\texttt{HTMLDoc}}(d')) \in T_{\texttt{Bool}}(X,\Sigma')}}{and(not(d' = \texttt{home}(w)), \texttt{inst}_{\texttt{HTMLDoc}}(d')) \in E_{\texttt{Bool}}(X,\Sigma,\mathcal{F}_{decl})}$$

where $not(d' = \texttt{home}(w)) \in T_{\texttt{Bool}}(X,\Sigma')$ has already been derived at the beginning of this section (for d instead of d').

The overall derivation shows that our migration routines are well-formed and well-typed.

Deriving Semantics

Here we demonstrate how the $\overset{mig}{\rightsquigarrow}$ relation works. For this purpose, we derive the semantics for $\texttt{migDirHTML}(t_d, t_w)$, where

$$\begin{aligned} t_d &\equiv \texttt{Dir}(\texttt{nextID}^3(\texttt{initID})\text{“}\texttt{source}\text{“}, \{\texttt{Dir}(\texttt{nextID}(\texttt{initID})\text{“}\texttt{calc05}\text{“}, \{\}, \{\})\}_s, \{\})\\ t_w &\equiv \texttt{Website}(...). \end{aligned}$$

Recall that t_d reflects parts of the directory structure of our running example (the directory “source” contains the directory “calc05”). The website t_w is not further specified as this is largely unimportant for the derivation. Yet suppose it is a simple constructor term $\texttt{Website}(...)$. It is thus evaluable by $\overset{t}{\rightsquigarrow}$ and causes no state changes. The initial migration sequence Δ equals $\langle\mathcal{A}\rangle$, where $\mathcal{A}$ is an arbitrary starting state $\mathcal{A}$. The derivation starts by applying the rule **Σ-terms 2**, where we denote $gen(\Sigma) \cup (\emptyset, \emptyset, \emptyset, \emptyset, \mathcal{F}_{decl})$ by Σ' and the function body for $\texttt{migDirHTML}$ by e_1.

$$\frac{\begin{array}{l}(\mathcal{A},\Delta,\eta,t_d) \overset{mig}{\rightsquigarrow} (\mathcal{A},\Delta,v_{source}) \\ (\mathcal{A},\Delta,\eta,t_w) \overset{mig}{\rightsquigarrow} (\mathcal{A},\Delta,v_w) \\ (\mathcal{A},\Delta,\{d \mapsto v_{source}, w \mapsto v_w\},e_1) \overset{mig}{\rightsquigarrow} (\mathcal{A}',\Delta',v^{new}_{source}) \end{array} \quad \begin{array}{l} \texttt{migDirHTML}(t_d,t_w) \in T_{\texttt{Dir}}(X,\Sigma') \\ \texttt{migDirHTML}(d,w) = e_1 \in FD_{ex} \end{array}}{(\mathcal{A},\Delta,\eta,\texttt{migDirHTML}(t_d,t_w)) \overset{mig}{\rightsquigarrow} (\mathcal{A}',\Delta',v^{new}_{source})}$$

The values v_{source}, v_w, and v^{new}_{source} denote the value of the "source" directory, the value of the website w, and the migrated "source" directory, respectively. v_{source} and v_w are derived using the rule **Σ-terms 1**, which results in evaluating the usual term semantics via $\overset{t}{\rightsquigarrow}$ as follows:

$$\frac{\begin{array}{l}\ldots \\ (\mathcal{A},\eta,t_d) \overset{t}{\rightsquigarrow} v_{source} \\ t_d \in T_{\texttt{Dir}}(X,gen(\Sigma))\end{array}}{(\mathcal{A},\Delta,\eta,t_d) \overset{mig}{\rightsquigarrow} (\mathcal{A},\Delta,v_{source})} \qquad \frac{\begin{array}{l}\ldots \\ (\mathcal{A},\eta,t_w) \overset{t}{\rightsquigarrow} v_w \\ t_w \in T_{\texttt{Website}}(X,gen(\Sigma))\end{array}}{(\mathcal{A},\Delta,\eta,t_w) \overset{mig}{\rightsquigarrow} (\mathcal{A},\Delta,v_w)}$$

Hence, we have to derive $(\mathcal{A},\Delta,\{d \mapsto v_{source}, w \mapsto v_w\},e_1) \overset{mig}{\rightsquigarrow} (\mathcal{A}',\Delta',v^{new}_{source})$. After applying the `let` rule

$$\frac{\begin{array}{l}\texttt{let } sDirs = \ldots \in E_{\texttt{Dir}}(X,\Sigma,\mathcal{F}_{decl}) \\ (\mathcal{A},\Delta,\quad \{d \mapsto v_{source}, w \mapsto v_w\}, \\ \qquad \{\texttt{migDirHTML}(d',w) \mid d' \leftarrow \ldots,\ \ldots\}) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2,\Delta_2,v_1) \\ (\mathcal{A}_2,\Delta_2,\{d \mapsto v_{source}, w \mapsto v_w, sDirs \mapsto v_1\}, \\ \qquad \{\texttt{migHTMLDoc}(d',w) \mid d' \leftarrow \ldots,\ \ldots\}) \overset{mig}{\rightsquigarrow} (\mathcal{A}_3,\Delta_3,v_2) \\ (\mathcal{A}_3,\Delta_3,\{d \mapsto v_{source}, w \mapsto v_w, sDirs \mapsto v_1, sDocs \mapsto v_2\}, \\ \qquad \texttt{transform}(d \mapsto (\texttt{Dir},\texttt{name}(d),sDirs,sDocs))) \overset{mig}{\rightsquigarrow} (\mathcal{A}',\Delta',v^{new}_{source})\end{array}}{\begin{array}{l}(\mathcal{A},\Delta,\{d \mapsto v_{source}, w \mapsto v_w\}, \\ \quad \texttt{let} \\ \quad\ sDirs = \{\texttt{migDirHTML}(d',w) \mid d' \leftarrow \ldots,\ \ldots\} \\ \quad\ sDocs = \{\texttt{migHTMLDoc}(d',w) \mid d' \leftarrow \ldots,\ \ldots\} \\ \quad \texttt{in transform}(d \mapsto (\texttt{Dir},\texttt{name}(d),sDirs,sDocs))) \overset{mig}{\rightsquigarrow} (\mathcal{A}',\Delta',v^{new}_{source})\end{array}}$$

we have three remaining derivation steps.

Step 1: Show

$$(\mathcal{A},\Delta,\ \{d \mapsto v_{source}, w \mapsto v_w\}, \quad \{\texttt{migDirHTML}(d',w) \mid d' \leftarrow \ldots,\ \ldots\}) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2,\Delta_2,v_1).$$

Step 2: Show

$$(\mathcal{A}_2,\Delta_2,\ \{d \mapsto v_{source}, w \mapsto v_w, sDirs \mapsto v_1\}, \quad \{\texttt{migHTMLDoc}(d',w) \mid d' \leftarrow \ldots,\ \ldots\}) \overset{mig}{\rightsquigarrow} (\mathcal{A}_3,\Delta_3,v_2).$$

Step 3: Show

$$(\mathcal{A}_3,\Delta_3,\ \{d \mapsto v_{source}, w \mapsto v_w, sDirs \mapsto v_1, sDocs \mapsto v_2\}, \quad \texttt{transform}(d \mapsto (\texttt{Dir},\texttt{name}(d),sDirs,sDocs))) \overset{mig}{\rightsquigarrow} (\mathcal{A}',\Delta',v^{new}_{source}).$$

Notice that the execution order is important since the single steps rely on previous results. We derive the semantics in consecutive order.

Step 1: We start by applying the **set comprehension** rule to

$\{\texttt{migDirHTML}(d',w) \mid d' \leftarrow \texttt{subDirs}(d),\ \texttt{True}\}) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2,\Delta_2,v_1)$.

This yields

$$\frac{\begin{array}{l}\{\texttt{migDirHTML}(d', w) \mid\ d' \leftarrow \texttt{subDirs}(d),\ \texttt{True}\} \in E_{\texttt{Set}[Dir]}(X, \Sigma, \mathcal{F}_{decl}) \\ y \notin \mathcal{FV}(e) \cup \mathcal{FV}(e') \cup \mathcal{FV}(e_B) \\ (\mathcal{A}, \Delta, \{d \mapsto v_{source}, w \mapsto v_w\}, \\ \quad \texttt{let}\ y = \texttt{subDirs}(d) \\ \quad \texttt{in}\ \ \texttt{if}\quad y = \{\}\ \texttt{then}\ \{\} \\ \quad\quad \texttt{else let}\ d' = \texttt{rep}(y)\ \texttt{in} \\ \quad\quad\quad \texttt{if True then}\ \{\texttt{migDirHTML}(d', w)\}_s \cup \\ \quad\quad\quad\quad\quad\quad \{\texttt{migDirHTML}(d', w) \mid\ d' \leftarrow y \backslash \{d'\}_s,\ \texttt{True}\} \\ \quad\quad\quad \texttt{else}\ \{\texttt{migDirHTML}(d', w) \mid\ d' \leftarrow y \backslash \{d'\}_s,\ \texttt{True}\}\ ) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2, \Delta_2, v_1)\end{array}}{\begin{array}{l}(\mathcal{A}, \Delta, \{d \mapsto v_{source}, w \mapsto v_w\}, \\ \quad \{\texttt{migDirHTML}(d', w) \mid\ d' \leftarrow \texttt{subDirs}(d),\ \texttt{True}\}\ ) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2, \Delta_2, v_1)\end{array}}$$

At this point it is important to notice that the expression $\texttt{subDirs}(d)$ is evaluated once upon first execution of the set comprehension term and then stored in the variable y. If $\texttt{subDirs}(d)$ caused state changes (which it does not in this example), these state changes would be performed only once. Therefore, we use the seemingly complicated variant with a `let` construct. The just-described effect becomes clearer with the following resolution of the `let` construct.

$$\frac{\begin{array}{l}(\mathcal{A}, \Delta, \{d \mapsto v_{source}, w \mapsto v_w\}, \texttt{subDirs}(d)) \overset{mig}{\rightsquigarrow} (\mathcal{A}, \eta, v_{sDirs}) \\ (\mathcal{A}, \Delta, \{d \mapsto v_{source}, w \mapsto v_w, y \mapsto v_{sDirs}\}, \\ \quad \texttt{if}\quad y = \{\}\ \texttt{then}\ \{\} \\ \quad \texttt{else let}\ d' = \texttt{rep}(y)\ \texttt{in} \\ \quad\quad \texttt{if True then}\ \{\texttt{migDirHTML}(d', w)\}_s \cup \\ \quad\quad\quad\quad\quad \{\texttt{migDirHTML}(d', w) \mid\ d' \leftarrow y \backslash \{d'\}_s,\ \texttt{True}\} \\ \quad\quad \texttt{else}\ \{\texttt{migDirHTML}(d', w) \mid\ d' \leftarrow y \backslash \{d'\}_s,\ \texttt{True}\}\ ) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2, \Delta_2, v_1)\end{array}}{\begin{array}{l}\{\texttt{migDirHTML}(d', w) \mid\ d' \leftarrow \texttt{subDirs}(d),\ \texttt{True}\} \in E_{\texttt{Set}[Dir]}(X, \Sigma, \mathcal{F}_{decl}) \\ (\mathcal{A}, \Delta, \{d \mapsto v_{source}, w \mapsto v_w\}, \\ \quad \texttt{let}\ y = \texttt{subDirs}(d) \\ \quad \texttt{in}\ \ \texttt{if}\quad y = \{\}\ \texttt{then}\ \{\} \\ \quad\quad \texttt{else let}\ d' = \texttt{rep}(y)\ \texttt{in} \\ \quad\quad\quad \texttt{if True then}\ \{\texttt{migDirHTML}(d', w)\}_s \cup \\ \quad\quad\quad\quad\quad\quad \{\texttt{migDirHTML}(d', w) \mid\ d' \leftarrow y \backslash \{d'\}_s,\ \texttt{True}\} \\ \quad\quad\quad \texttt{else}\ \{\texttt{migDirHTML}(d', w) \mid\ d' \leftarrow y \backslash \{d'\}_s,\ \texttt{True}\}\ ) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2, \Delta_2, v_1)\end{array}}$$

Now, the upper variable assignment assigns the value v_{sDirs} to y; this value represents the set $\{\texttt{Dir}(\text{“}\texttt{calc05}\text{“}, ...)\}_s$. The assignment for this value is passed through to the recursive set comprehension terms. Hence, the source set need not be computed again.

We slightly abbreviate the further derivation steps. First, the value v_{sDirs} is the evaluation result of the simple function call $\texttt{subDirs}(d)$ and, hence, is determined by $\overset{t}{\rightsquigarrow}$. Also, we abbreviate evaluation of the `if` clauses. At this point $y \neq \{\}$ and `True` hold. Therefore, the semantics of the above expression is determined by evaluating the expression

$$\begin{array}{l}\texttt{let}\ d' = \texttt{rep}(y) \\ \texttt{in}\ \ \{\texttt{migDirHTML}(d', w)\}_s \cup \{\texttt{migDirHTML}(d', w) \mid\ d' \leftarrow y \backslash \{d'\}_s,\ \texttt{True}\}.\end{array}$$

The `let` construct assigns a value $v_{calc} := \texttt{rep}^{\mathcal{A}}(y) = \texttt{rep}^{\mathcal{A}}(v_{sDirs})$ to d'. We call this value v_{calc} since it represents the “calc05” directory in our example. Notice the role of the function `rep`. It chooses a representative of a set. We fix the evaluation order of set comprehension in this way. Yet we leave it up to the user to specify this function in detail.

The current variable assignment is $\{d \mapsto v_{source}, w \mapsto v_w, y \mapsto v_{sDirs}, d' \mapsto v_{calc}\}$. Since $y \backslash \{d'\}$ evaluates to the empty set $\{\}$ under the given variable assignment (the directory "source" contains the directory "calc05", only), the expression

$$\{\texttt{migDirHTML}(d', w) \mid d' \leftarrow y \backslash \{d'\}_s,\ \texttt{True}\}$$

evaluates to $\{\}$. Hence, we only have to derive the semantics for $\{\texttt{migDirHTML}(d', w)\}_s$ which results in the following derivation tree (rule **Σ-terms 3** and **Σ-terms 2** are applied consecutively):

$$\dfrac{\begin{array}{l}(\mathcal{A}, \Delta, \{d \mapsto v_{source}, w \mapsto v_w, y \mapsto v_{sDirs}, d' \mapsto v_{calc}\},\\ \quad d') \overset{mig}{\rightsquigarrow} (\mathcal{A}, \Delta, v_{calc})\\ (\mathcal{A}, \Delta, \{d \mapsto v_{source}, w \mapsto v_w, y \mapsto v_{sDirs}, d' \mapsto v_{calc}\},\\ \quad w) \overset{mig}{\rightsquigarrow} (\mathcal{A}, \Delta, v_w)\\ (\mathcal{A}, \Delta, \{d \mapsto v_{calc}, w \mapsto v_w, y \mapsto v_{sDirs}, d' \mapsto v_{calc}\},\\ \quad e_1) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2, \Delta_2, v^{new}_{calc})\end{array} \quad \begin{array}{l}\texttt{migDirHTML}(d', w) \in T_{\texttt{Dir}}(X, \Sigma')\\ \texttt{migDirHTML}(d, w) = e_1 \in FD_{ex}\end{array}}{\dfrac{\begin{array}{c}(\mathcal{A}, \Delta, \{d \mapsto v_{source}, w \mapsto v_w, y \mapsto v_{sDirs}, d' \mapsto v_{calc}\},\\ \texttt{migDirHTML}(d', w)) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2, \Delta_2, v^{new}_{calc})\end{array}}{\begin{array}{c}\{\}^{\mathcal{A}}_s(v^{new}_{calc}) = v_1\\ (\mathcal{A}, \Delta, \{d \mapsto v_{source}, w \mapsto v_w, y \mapsto v_{sDirs}, d' \mapsto v_{calc}\},\\ \{\texttt{migDirHTML}(d', w)\}_s) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2, \Delta_2, v_1)\end{array}}}$$

There d' and w are evaluated to $v_{calc} = \eta(d')$ and $v_w = \eta(w)$, respectively. In the third expression v^{new}_{calc} represents the value of the migrated directory. Notice the variable assignment in the derivation for e_1. According to rule **Σ-terms 2**, d is bound to the value v_{calc}. Recall that v_{calc} is the value of the representative of $\texttt{subDirs}(d)$ (above), which is the semantic value of the "calc05" directory in our example. Since "calc05" does neither have sub-directories nor sub-documents, both set comprehension expressions in the body of `migDirHTML` evaluate to $\{\}$, which yields the variable assignments $sDirs \mapsto \emptyset$ and $sDocs \mapsto \emptyset$. Hence, the derivation step for $(\mathcal{A}, \Delta, \{d \mapsto v_{calc}, w \mapsto v_w, y \mapsto v_{sDirs}, d' \mapsto v_{calc}\}, e_1)$ corresponds to deriving the semantics for $(\mathcal{A}, \Delta, \{d \mapsto v_{calc}, w \mapsto v_w, y \mapsto \emptyset, d' \mapsto v_{calc}, sDirs \mapsto \emptyset, sDocs \mapsto \emptyset\}, \texttt{transform}(d \mapsto (\texttt{Dir}, ...)))$.

This derivation uses the rule **Basic operations** and starts as follows:

$$\dfrac{\texttt{transform}(d \mapsto (\texttt{Dir}, ...)) \in \mathcal{BO}p_\tau(X, gen(\Sigma)) \qquad \dfrac{\ldots \qquad \ldots}{\begin{array}{l}(\mathcal{A}, \Delta, \{d \mapsto v_{calc}, w \mapsto v_w, y \mapsto v_{sDirs}, d' \mapsto v_{calc},\\ \quad sDirs \mapsto \emptyset, sDocs \mapsto \emptyset\},\\ \quad \texttt{transform}(d \mapsto (\texttt{Dir}, ...))) \overset{op}{\rightsquigarrow} (\mathcal{A}', \Delta', v^{new}_{source})\end{array}}}{\begin{array}{c}(\mathcal{A}, \Delta, \{d \mapsto v_{calc}, w \mapsto v_w, y \mapsto v_{sDirs}, d' \mapsto v_{calc}, sDirs \mapsto \emptyset, sDocs \mapsto \emptyset\},\\ \texttt{transform}(d \mapsto (\texttt{Dir}, \texttt{name}(d), sDirs, sDocs))) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v^{new}_{source})\end{array}}$$

We proceed by applying the rule **Object transformation 2** for $\overset{op}{\rightsquigarrow}$ (cf. Tab. 6.5, page 95).

$$\frac{\begin{array}{l}\texttt{transform}(d \mapsto (\texttt{Dir}, \texttt{name}(d), sDirs, sDocs)) \in \mathcal{BO}p_{\texttt{Dir}}(X, gen(\Sigma)) \\ [\![r\_transform(d, \texttt{Dir}, (\texttt{name}(d), sDirs, sDocs), \{d\})]\!]^{\mathcal{A}}_{\{d \mapsto v_{calc}, sDirs \mapsto \emptyset, sDocs \mapsto \emptyset\}} = U \\ U \neq \emptyset \\ (\mathcal{A}, \Delta, \eta, \texttt{Dir}(\texttt{oidToUse}, \texttt{name}(d), sDirs, sDocs)) \overset{t}{\rightsquigarrow} v^{new}_{calc} \\ ct_{calc} \in CT^{min}_{\mathcal{A}}(\Sigma, v_{calc}) \\ ct^{new}_{calc} \in CT^{min}_{\mathcal{A}}(\Sigma, v^{new}_{calc})\end{array}}{\begin{array}{l}(\mathcal{A}, \Delta, \{d \mapsto v_{calc}, w \mapsto v_w, y \mapsto v_{sDirs}, d' \mapsto v_{calc}, sDirs \mapsto \emptyset, sDocs \mapsto \emptyset\}, \\ \quad \texttt{transform}(d \mapsto (\texttt{Dir}, \texttt{name}(d), sDirs, sDocs))) \overset{op}{\rightsquigarrow} (\ \mathcal{A} + U, \\ \qquad\qquad\qquad\qquad \Delta; \langle \mathcal{A}, \mathsf{trans}(ct_{calc} \mapsto ct^{new}_{calc}), \mathcal{A} + U \rangle, \\ \qquad\qquad\qquad\qquad v^{new}_{calc})\end{array}}$$

This can be derived under the assumption that $\mathcal{A} \models d \in \texttt{existDOObj}[\emptyset[d \mapsto v_{calc}]]$. Since v_{calc} is the value of the "calc05" directory in our example, the assumption holds. Thus step one is completed. The overall derivation has delivered the following:

$$(\mathcal{A}, \Delta,\ \{d \mapsto v_{source}, w \mapsto v_w\},\ \{\texttt{migDirHTML}(d', w) \mid\ d' \leftarrow \texttt{subDirs}(d),\ \texttt{True}\}\ ) \overset{mig}{\rightsquigarrow} (\mathcal{A}_2, \Delta_2, v_1)$$

where

$$\begin{array}{lcl}\mathcal{A}_2 & = & \mathcal{A} + U \\ v_{calc} & = & \mathcal{V}^{\mathcal{A}}[\![\texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\})]\!] \\ v^{new}_{calc} & = & \mathcal{V}^{\mathcal{A}}[\![\texttt{Dir}(\texttt{oidToUse}, \text{“}\texttt{calc05}\text{“}, \{\}, \{\})]\!] \\ \Delta_2 & = & \Delta; \langle \mathcal{A}, \mathsf{trans}(ct_{calc} \mapsto ct^{new}_{calc}), \mathcal{A} + U \rangle \\ v_1 & = & \{v^{new}_{calc}\}^{\mathcal{A}}_{s} \ = \ \mathcal{V}^{\mathcal{A}}[\![\{\texttt{Dir}(\texttt{oidToUse}, \text{“}\texttt{calc05}\text{“}, \{\}, \{\})\}_s]\!]\end{array}$$

At this point, the derivation yields an element of type $\texttt{Set}[\texttt{Dir}]$ that contains the transformation result for the directory "calc05". We proceed with step two.

Step 2: This step computes the derivation

$$(\mathcal{A}_2, \Delta_2,\ \{d \mapsto v_{source}, w \mapsto v_w, sDirs \mapsto v_1\}\ \{\texttt{migHTMLDoc}(d', w) \mid\ d' \leftarrow \texttt{subDocs}(d),\ and(not(d' = \texttt{home}(w)), \texttt{inst}_{\texttt{HTMLDoc}}(d'))\}\ ) \overset{mig}{\rightsquigarrow} (\mathcal{A}_3, \Delta_3, v_2)$$

Since $\texttt{subDocs}(d)$ is empty (the directory "source" contains no sub documents), $\mathcal{A}_3 = \mathcal{A}_2$, $\Delta_3 = \Delta_2$, and $v_2 = \{\}^{\mathcal{A}}$ holds. Hence, we can directly switch to the last step.

Step 3: The remaining derivation target looks as follows:

$$(\mathcal{A}_3, \Delta_3,\ \{d \mapsto v_{source}, w \mapsto v_w, sDirs \mapsto v_1, sDocs \mapsto v_2\},\ \texttt{transform}(d \mapsto (\texttt{Dir}, \texttt{name}(d), sDirs, sDocs))) \overset{mig}{\rightsquigarrow} (\mathcal{A}', \Delta', v^{new}_{source})$$

Since it is very similar to the transformation operation just shown, we omit a detailed derivation. Yet this step generates the overall result for the initial function call $\texttt{migDirHTML}(t_d, t_w)$. Due to the variable assignment $d \mapsto v_{source}$ (v_{source} represents the directory "source"), the directory "source" is transformed. The target directory receives the name "source" as well. The sub-directories and sub documents of the transformation target are set to $sDirs$ and $sDocs$, respectively. The latter two have been computed in steps one and two. The values are bound to the respective variables in the variable assignment ($sDirs \mapsto v_1, sDocs \mapsto v_2$). The derivation for this operation constructs another update set U', such that $\mathcal{A}' = \mathcal{A} + U + U'$. The value v represents the transformation result and Δ_3 is extended by $\mathsf{trans}(ct_{source} \mapsto ct^{new}_{source})$, where ct_{source} and ct^{new}_{source} are minimal constructor terms for v_{source} and v^{new}_{source}, respectively. Hence, the overall derivation yields

$$(\mathcal{A}, \Delta, \texttt{migDirHTML}(t_d, t_w)) \overset{mig}{\rightsquigarrow} (\ \mathcal{A} + U + U', \Delta; \langle \mathcal{A}, \mathsf{trans}(ct_{calc} \mapsto ct^{new}_{calc}), \mathcal{A} + U \rangle; \langle \mathcal{A} + U, \mathsf{trans}(ct_{source} \mapsto ct^{new}_{source}), \mathcal{A} + U + U' \rangle, v^{new}_{source})$$

where

$$
\begin{array}{lll}
t_d & \equiv & \texttt{Dir}(\texttt{nextID}^3(\texttt{initID}), \text{“}\texttt{source}\text{“}, \{\texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\})\}_s, \{\}), \\
v_{source} & = & \mathcal{V}^{\mathcal{A}}[\![t_d]\!], \\
v_{calc} & = & \mathcal{V}^{\mathcal{A}}[\![\texttt{Dir}(\texttt{nextID}(\texttt{initID}), \text{“}\texttt{calc05}\text{“}, \{\}, \{\})]\!], \\
v_{calc}^{new} & = & \mathcal{V}^{\mathcal{A}}[\![\texttt{Dir}(\texttt{oidToUse}, \text{“}\texttt{calc05}\text{“}, \{\}, \{\})]\!], \text{ and} \\
v_{source}^{new} & = & \mathcal{V}^{\mathcal{A}+U}[\![\texttt{Dir}(\texttt{oidToUse}, \text{“}\texttt{source}\text{“}, \{x\}_s, \{\})]\!]\{x \mapsto v_{calc}^{new}\}
\end{array}
$$

The semantics confirms the behavior of $\texttt{migDirHTML}(d, w)$ as explained in the introduction to Chap. 6. This function first transforms all sub-directories and sub-documents of d and collects all these results in sets. Then it transforms the directory d to a new representation and attaches the transformed sub directories and sub-documents. Hence, it corresponds to a kind of recursive cloning of d.

Appendix C

Proofs

The proofs are structured by chapter.

C.1 Proofs for Chap. 3

Proof C.1.1 (Proof of Lemma 3.2.2) We show the properties of Defn. 3.2.5 consecutively.

(1) $t \in T_{\tau'}(X,\Sigma) \wedge \tau' < \tau \Rightarrow t \in T_\tau(X,\Sigma)$ according to rule **Subtyping** in Tab. 3.2.

(2) We show the property by induction on the structure of t.

Case 1 ($t \equiv x$): We conclude:

$$
\begin{array}{lll}
x \in T_\tau(X,\Sigma) & \Rightarrow \exists x : \tau' \in X_{\tau'} \bullet \tau' < \tau \wedge (\mathcal{A},\eta,x) \overset{t}{\rightsquigarrow} \eta(x) & \text{Tab. 3.2} \\
& \Rightarrow \tau' < \tau \wedge (\mathcal{A},\eta,x) \overset{t}{\rightsquigarrow} \eta(x) \wedge \eta(x) \in \tau'^{\mathcal{A}} \cup \{\bot\} & \text{Defn. 3.2.4} \\
& \Rightarrow \tau'^{\mathcal{A}} \subseteq \tau^{\mathcal{A}} \wedge (\mathcal{A},\eta,x) \overset{t}{\rightsquigarrow} \eta(x) \wedge \eta(x) \in \tau^{\mathcal{A}} \cup \{\bot\} & \text{Defn. 3.2.3} \\
& \Rightarrow (\mathcal{A},\eta,x) \overset{t}{\rightsquigarrow} \eta(x) \wedge \eta(x) \in \tau^{\mathcal{A}} \cup \{\bot\} & \\
& \Rightarrow \exists v \in \tau^{\mathcal{A}} \cup \{\bot\} \bullet (\mathcal{A},\eta,x) \overset{t}{\rightsquigarrow} v &
\end{array}
$$

Case 2 ($t \equiv f(\overline{t_i})$): We conclude:

$$
\begin{array}{ll}
f(\overline{t_i}) \in T_\tau(X,\Sigma) & \\
\Rightarrow \exists f_{\overline{\tau_i}\to\tau'} \in \mathcal{F}_{\overline{\tau_i}\to\tau'} \bullet \tau' < \tau \wedge \overline{t_i} \in \overline{T_{\tau_i}(X,\Sigma)} & \text{Tab. 3.2} \\
\Rightarrow \exists f_{\overline{\tau_i}\to\tau'} \in \mathcal{F}_{\overline{\tau_i}\to\tau'} \bullet \tau' < \tau \wedge \overline{t_i} \in \overline{T_{\tau_i}(X,\Sigma)} \wedge & \\
\quad \exists \overline{v_i} \in \overline{\tau_i^{\mathcal{A}} \cup \{\bot\}} \bullet (\mathcal{A},\eta,t_1) \overset{t}{\rightsquigarrow} v_1 \wedge \ldots \wedge (\mathcal{A},\eta,t_n) \overset{t}{\rightsquigarrow} v_n & \text{ind.hyp.} \\
\Rightarrow \tau' < \tau \wedge (\mathcal{A},\eta,f(\overline{t_i})) \overset{t}{\rightsquigarrow} f^{\mathcal{A}^\bot}(\overline{v_i}) \wedge f^{\mathcal{A}^\bot}(\overline{v_i}) \in \tau'^{\mathcal{A}} \cup \{\bot\} & \text{Tab. 3.2, Defn. 3.2.3} \\
\Rightarrow \tau' < \tau \wedge \exists v \in \tau'^{\mathcal{A}} \cup \{\bot\} \bullet (\mathcal{A},\eta,f(\overline{t_i})) \overset{t}{\rightsquigarrow} v & \\
\Rightarrow \tau'^{\mathcal{A}} \subseteq \tau^{\mathcal{A}} \wedge \exists v \in \tau'^{\mathcal{A}} \cup \{\bot\} \bullet (\mathcal{A},\eta,f(\overline{t_i})) \overset{t}{\rightsquigarrow} v & \text{Defn. 3.2.3} \\
\Rightarrow \exists v \in \tau^{\mathcal{A}} \cup \{\bot\} \bullet (\mathcal{A},\eta,x) \overset{t}{\rightsquigarrow} v &
\end{array}
$$

□

Proof C.1.2 (Proof of Lemma 3.2.3) We show the property by induction on the structure of t. Ambiguity can arise due to symbol overloading only. Therefore, we start the proofs by assuming two different typings for t.

Case 1 ($t \equiv x$): Informally, the property holds since $X_{\tau'} \cap X_\tau = \emptyset$ if $\tau \neq \tau'$ according to Defn. 3.2.4 and, hence, $\eta(x)$ is uniquely determined. Formally, we conclude:

$x \in T_\tau(X,\Sigma) \wedge (\mathcal{A},\eta,x) \overset{t}{\rightsquigarrow} v_1 \wedge (\mathcal{A},\eta,x) \overset{t}{\rightsquigarrow} v_2$
$\Rightarrow \exists x:\tau_1' \in X_{\tau_1'}, x:\tau_2' \in X_{\tau_2'} \bullet v_1 = \eta(x:\tau_1') \wedge v_2 = \eta(x:\tau_2')$ Tab. 3.2
$\Rightarrow \tau_1' = \tau_2' \wedge v_1 = \eta(x:\tau_1') \wedge v_2 = \eta(x:\tau_2')$ Defn. 3.2.4
$\Rightarrow v_1 = v_2$

Case 2 ($t \equiv f(\overline{t_i})$): In the proof we suppose $v_1 \neq \bot \vee v_2 \neq \bot$ and refer to this fact by (*). If both v_1 and v_2 are $\bot$, the assumption trivially holds.

$f(\overline{t_i}) \in T_\tau(X,\Sigma) \wedge (\mathcal{A},\eta,f(\overline{t_i})) \overset{t}{\rightsquigarrow} v_1 \wedge (\mathcal{A},\eta,f(\overline{t_i})) \overset{t}{\rightsquigarrow} v_2$
$\Rightarrow \exists f_{\overline{\tau_{1,i}}\to\tau_1'} \in \mathcal{F}_{\overline{\tau_{1,i}}\to\tau_1'}, f_{\overline{\tau_{2,i}}\to\tau_2'} \in \mathcal{F}_{\overline{\tau_{2,i}}\to\tau_2'} \bullet$
$(\mathcal{A},\eta,t_1) \overset{t}{\rightsquigarrow} w_1 \wedge ... \wedge (\mathcal{A},t_n,\eta) \overset{t}{\rightsquigarrow} w_n \wedge v_1 = f^{\mathcal{A}^\bot}_{\overline{\tau_{1,i}}\to\tau_1'}(\overline{w_i}) \wedge$
$(\mathcal{A},\eta,t_1) \overset{t}{\rightsquigarrow} w_1' \wedge ... \wedge (\mathcal{A},t_n,\eta) \overset{t}{\rightsquigarrow} w_n' \wedge v_2 = f^{\mathcal{A}^\bot}_{\overline{\tau_{2,i}}\to\tau_2'}(\overline{w_i'})$ Tab. 3.2
$\Rightarrow w_1 = w_1' \wedge ... \wedge w_n = w_n' \wedge$ ind.hyp.
$v_1 = f^{\mathcal{A}^\bot}_{\overline{\tau_{1,i}}\to\tau_1'}(\overline{w_i}) \ \wedge \ v_2 = f^{\mathcal{A}^\bot}_{\overline{\tau_{2,i}}\to\tau_2'}(\overline{w_i}) \wedge$
$\overline{w_i} \in \overline{\tau_{1,i}^{\mathcal{A}}} \wedge \overline{w_i} \in \overline{\tau_{2,i}^{\mathcal{A}}}$ (*), Lemma 3.2.2(2)
$\Rightarrow v_1 = f^{\mathcal{A}^\bot}_{\overline{\tau_{1,i}}\to\tau_1'}(\overline{w_i}) \ \wedge \ v_2 = f^{\mathcal{A}^\bot}_{\overline{\tau_{2,i}}\to\tau_2'}(\overline{w_i}) \wedge$
$\overline{w_i} \in \overline{\tau_{1,i}^{\mathcal{A}}} \wedge \overline{w_i} \in \overline{\tau_{2,i}^{\mathcal{A}}}$
$\exists \overline{\tau_j} \bullet (\overline{\tau_j}<\overline{\tau_{1,i}} \wedge \overline{\tau_j}<\overline{\tau_{2,i}}) \wedge \tau_1', \tau_2' < \texttt{Top}$ Defn. 3.2.3
$\Rightarrow v_1 = f^{\mathcal{A}^\bot}_{\overline{\tau_{1,i}}\to\tau_1'}(\overline{w_i}) \ \wedge \ v_2 = f^{\mathcal{A}^\bot}_{\overline{\tau_{2,i}}\to\tau_2'}(\overline{w_i}) \wedge$
$\overline{w_i} \in \overline{\tau_{1,i}^{\mathcal{A}}} \wedge \overline{w_i} \in \overline{\tau_{2,i}^{\mathcal{A}}} \wedge$
$f_{\overline{\tau_{1,i}}\to\tau_1'} \cong_{\mathcal{F}} f_{\overline{\tau_{1,i}}\to\tau_1'}$ Defn. 3.2.2
$\Rightarrow v_1 = v_2$ Defn. 3.2.3

□

Proof C.1.3 (Proof of Lemma 3.3.1) As $<_{id}$ is transitive and irreflexive, the property directly follows. □

Proof C.1.4 (Proof of Lemma 3.3.2) We start by proving parts (1) and (2) of the ID-property.

(1) In Σ_s^{DA} `DObj` is the only subtype of `DObj`. It has one constructor only (`createDObj`), which has the required parameter of type `OID` in the first position.

(2) The specification for type `DObj` contains the axiom

$$\forall id : \texttt{OID} \bullet \texttt{oid}(\texttt{createDObj}(id)) = id.$$

The property, thus, holds.

We proceed by proving parts (1) and (2) of the constructor property. In both proofs we will use the following property: Whenever $\mathcal{A} \models t \in s$, $t \in GT_\tau(\Sigma), s \in GT_{\texttt{Set[Top]}}(\Sigma)$, there is $t' \in T_\tau^{\leq}(\emptyset, s)$ such that $\mathcal{A} \models t = t'$. We prove this by induction on the structure of s. Notice that the set constructors and the function `rep` are the only operations that return elements of type `Set`. Also, $\mathcal{A} \models t \in s$ excludes $s \equiv \{\}$ according to Defn. 3.3.1.

Case 1 ($s \equiv \{t'\}_s$): Then $t' \in T_\tau^{\leq}(\emptyset, s)$ and $\mathcal{A} \models t \in s \Leftrightarrow t = t'$ according to Defn. 3.3.1.

Case 2 ($s \equiv s' \cup s''$): According to Defn. 3.3.1 $\mathcal{A} \models t \in s \Leftrightarrow t \in s' \vee t \in s''$. By induction hypothesis the property follows.

Case 3 ($s \equiv \texttt{rep}(s')$): We conclude

$s \equiv \texttt{rep}(s') \wedge \mathcal{A} \models t \in s$
$\Rightarrow s \equiv \texttt{rep}(s') \wedge \mathcal{A} \models t \in s \wedge s \in s'$ Defn. 3.3.1
$\Rightarrow s \equiv \texttt{rep}(s') \wedge \mathcal{A} \models t \in s \wedge s \in s' \wedge$
there is $s'' \in T^{\leq}_{\texttt{Top}}(\emptyset, s')$ such that $\mathcal{A} \models s = s''$ ind.hyp.
$\Rightarrow s \equiv \texttt{rep}(s') \wedge \mathcal{A} \models t \in s \wedge s \in s' \wedge$ Defn. 3.3.1
there is $s'' \in T^{\leq}_{\texttt{Top}}(\emptyset, s')$ such that $\mathcal{A} \models t \in s''$
$\Rightarrow s \equiv \texttt{rep}(s') \wedge \mathcal{A} \models t \in s \wedge s \in s' \wedge$
there is $s'' \in T^{\leq}_{\texttt{Top}}(\emptyset, s')$ such that $\mathcal{A} \models t \in s'' \wedge$
there is $t' \in T^{\leq}_{\tau}(\emptyset, s'')$ such that $\mathcal{A} \models t = t'$ ind.hyp.
$\Rightarrow s \equiv \texttt{rep}(s') \wedge \mathcal{A} \models t \in s \wedge s \in s' \wedge$
there is $t' \in T^{\leq}_{\tau}(\emptyset, s')$ such that $\mathcal{A} \models t = t'$
$\Rightarrow$ there is $t' \in T^{\leq}_{\tau}(\emptyset, s)$ such that $\mathcal{A} \models t = t'$

Now, we are ready to prove that $Spec^{DA}_s$ satisfies both constructor properties.

(1) We prove the property by induction on the structure of t. Notice that the constructor `createDObj` and the function `rep` applied to a term of type `Set`[`DObj`] are the only functions of Σ^{DA}_s that return a result of type `DObj`.

Case 1 ($t \equiv \texttt{createDObj}(\overline{t'_i})$): Since `createDObj` is a constructor and $\texttt{createDObj}(\overline{t'_i}) \in T^{\leq}_{\texttt{DObj}}(\emptyset, \texttt{createDObj}(\overline{t'_i}))$, $\texttt{createDObj}(\overline{t'_i})$ satisfies the property.

Case 2 ($t \equiv \texttt{rep}(s)$): We conclude

$t \equiv \texttt{rep}(s) \Rightarrow \mathcal{A} \models t \in s$ Defn. 3.3.1
$\Rightarrow$ there is $t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, s)$ such that $\mathcal{A} \models t = t'$ above
$\Rightarrow$ there is $t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, s)$ such that $\mathcal{A} \models t = t' \wedge$
there is $c \in \mathcal{C}, c(\overline{t_i}) \in T^{\leq}_{\texttt{DObj}}(\emptyset, t')$ such that $\mathcal{A} \models c(\overline{t_i}) = t$ ind.hyp.
$\Rightarrow$ there is $c \in \mathcal{C}, c(\overline{t_i}) \in T^{\leq}_{\texttt{DObj}}(\emptyset, s)$ such that $\mathcal{A} \models c(\overline{t_i}) = t$
$\Rightarrow$ there is $c \in \mathcal{C}, c(\overline{t_i}) \in T^{\leq}_{\texttt{DObj}}(\emptyset, t)$ such that $\mathcal{A} \models c(\overline{t_i}) = t$

(2) Given two minimal constructor terms ct, ct' of type `Set`[τ] such that $\mathcal{A} \models ct = ct'$, we conclude

$\mathcal{A} \models ct = ct' \Leftrightarrow \mathcal{A} \models \forall x : \tau \bullet x \in ct \Leftrightarrow x \in ct'$ Defn. 3.3.1
$\Rightarrow \mathcal{A} \models \forall x : \texttt{DObj} \bullet x \in ct \Leftrightarrow x \in ct'$
$\Rightarrow$ for all $t \in GT_{\texttt{DObj}}(\Sigma)$ it is true that
(there is $t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct)$ such that $\mathcal{A} \models t = t') \Leftrightarrow$ above,
(there is $t'' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct')$ such that $\mathcal{A} \models t = t''$) Defn. 3.2.11
$\Rightarrow$ for all $t \in GT_{\texttt{DObj}}(\Sigma)$ it is true that
$(\mathcal{V}^{\mathcal{A}}[\![t]\!] \in \{\mathcal{V}^{\mathcal{A}}[\![t']\!] \mid t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct)\}) \Leftrightarrow$ Defn. 3.2.11
$(\mathcal{V}^{\mathcal{A}}[\![t]\!] \in \{\mathcal{V}^{\mathcal{A}}[\![t'']\!] \mid t'' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct')\})$
$\Rightarrow \{\mathcal{V}^{\mathcal{A}}[\![t']\!] \mid t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct)\} = \{\mathcal{V}^{\mathcal{A}}[\![t'']\!] \mid t'' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct')\}$

□

Proof C.1.5 (Proof of Lemma 3.3.3) We prove the properties in consecutive order.

(1) We conclude

$\mathcal{A} \models s \in cont^{\mathcal{A},\eta}_{\texttt{DObj}}(t)[\eta]$
$\Leftrightarrow$ there is a term $t' \in cont^{\mathcal{A},\eta}_{\texttt{DObj}}(t)$ such that $\mathcal{A} \models s = t'[\eta]$ Defn. 3.3.3
$\Leftrightarrow$ there is a term $t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct_t), ct_t \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta)$,
such that $\mathcal{A} \models s = t'[\eta]$ Defn. 3.3.3
$\Leftrightarrow \mathcal{V}^{\mathcal{A}}[\![s]\!]\eta \in \{\mathcal{V}^{\mathcal{A}}[\![t']\!]\eta | t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct_t)\}$ Defn. 3.2.11
$\Leftrightarrow$ there is $ct_t \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta)$ Defn. 3.3.2,
such that $\mathcal{V}^{\mathcal{A}}[\![s]\!]\eta \in \{\mathcal{V}^{\mathcal{A}}[\![t']\!]\eta \mid t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct_t)\}$ constr.prop. (2)

(2) We have to show reflexivity, transitivity, and antisymmetry.

Reflexivity: Given a term $t \in GT_{\mathtt{DObj}}(\Sigma)$, we conclude

$t \in GT_{\mathtt{DObj}}(\Sigma)$
$\Rightarrow t \in GT_{\mathtt{DObj}}(\Sigma) \wedge$ for all $ct \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!])\ \mathcal{A} \models t = ct$ Defn. 3.2.9
$\Rightarrow t \in GT_{\mathtt{DObj}}(\Sigma) \wedge$ for all $ct \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!])\ \mathcal{A} \models t = ct$
and $ct \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct)$ Defn. 3.2.6
$\Rightarrow t \in GT_{\mathtt{DObj}}(\Sigma) \wedge$
there is a term $t' \in cont^{\mathcal{A}}_{\mathtt{DObj}}(t)$ such that $\mathcal{A} \models t = t'$ Defn. 3.3.3, (*)
$\Rightarrow t \in GT_{\mathtt{DObj}}(\Sigma) \wedge \mathcal{A} \models t \in cont^{\mathcal{A}}_{\mathtt{DObj}}(t)$ Defn. 3.3.3
$\Rightarrow (t,t) \in R^{cont}_{\mathcal{A}}$

With (*) we refer to the fact that $t' = ct_t$ satisfies the condition, where ct_t is the fixed minimal constructor term of Defn. 3.3.3.

Antisymmetry: We conclude

$(s,t) \in R^{cont}_{\mathcal{A}} \wedge (t,s) \in R^{cont}_{\mathcal{A}}$
$\Rightarrow \mathcal{A} \models s \in cont^{\mathcal{A}}_{\mathtt{DObj}}(t) \wedge \mathcal{A} \models t \in cont^{\mathcal{A}}_{\mathtt{DObj}}(s)$
$\Rightarrow$ there is a term $t' \in cont^{\mathcal{A}}_{\mathtt{DObj}}(t)$ such that $\mathcal{A} \models s = t' \wedge$
there is a term $s' \in cont^{\mathcal{A}}_{\mathtt{DObj}}(s)$ such that $\mathcal{A} \models t = s'$ Defn. 3.3.3
$\Rightarrow$ there are terms t', ct_t such that $t' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_t)$,
$ct_t \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]), \mathcal{A} \models s = t' \wedge ct_t = t$, and
there are terms s', ct_s such that $s' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_s)$,
$ct_s \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![s]\!]), \mathcal{A} \models t = s' \wedge ct_s = s$ Defn. 3.3.3
$\Rightarrow \mathcal{A} \models t' = ct_s \wedge s = ct_s \wedge l(t') = l(ct_s) \wedge t' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_t) \wedge$ t' min.constr.for s
$\mathcal{A} \models s' = ct_t \wedge t = ct_t \wedge l(s') = l(ct_t) \wedge s' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_s)$ s' min.constr.for t
$\Rightarrow \mathcal{A} \models t' = ct_s \wedge s = ct_s \wedge l(t') = l(ct_s) \wedge l(t') \leq l(ct_t) \wedge$
$\mathcal{A} \models s' = ct_t \wedge t = ct_t \wedge l(s') = l(ct_t) \wedge l(s') \leq l(ct_s)$
$\Rightarrow \mathcal{A} \models t' = ct_s \wedge s = ct_s \wedge l(t') \leq l(s') \wedge$
$\mathcal{A} \models s' = ct_t \wedge t = ct_t \wedge l(s') \leq l(t')$
$\Rightarrow \mathcal{A} \models t' = ct_s \wedge s = ct_s \wedge l(t') = l(s') \wedge$
$\mathcal{A} \models s' = ct_t \wedge t = ct_t$
$\Rightarrow \mathcal{A} \models t' = s \wedge t' = s' \wedge s' = t$ $t' \equiv s'$
$\Rightarrow \mathcal{A} \models s = t$

Transitivity: We conclude

$(s,t) \in R^{cont}_{\mathcal{A}} \wedge (t,u) \in R^{cont}_{\mathcal{A}}$
$\Rightarrow \mathcal{A} \models s \in cont^{\mathcal{A}}_{\mathtt{DObj}}(t) \wedge \mathcal{A} \models t \in cont^{\mathcal{A}}_{\mathtt{DObj}}(u)$
$\Rightarrow$ there is $ct_t \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!])$ such that
$\mathcal{V}^{\mathcal{A}}[\![s]\!] \in \{\mathcal{V}^{\mathcal{A}}[\![t']\!] \mid t' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_t)\}$ and
$\Rightarrow$ there is $ct_u \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![u]\!])$ such that
$\mathcal{V}^{\mathcal{A}}[\![t]\!] \in \{\mathcal{V}^{\mathcal{A}}[\![u']\!] \mid u' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_u)\}$ Lemma 3.3.3(1)
$\Rightarrow \mathcal{V}^{\mathcal{A}}[\![s]\!] \in \{\mathcal{V}^{\mathcal{A}}[\![t']\!] \mid t' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_t)\} \wedge$
there is $u_t \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_u)$ such that $\mathcal{V}^{\mathcal{A}}[\![t]\!] = \mathcal{V}^{\mathcal{A}}[\![u_t]\!] \wedge$
$u_t, c_t \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!])$ ct_u min.constr.
$\Rightarrow \mathcal{V}^{\mathcal{A}}[\![s]\!] \in \{\mathcal{V}^{\mathcal{A}}[\![t']\!] \mid t' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_t)\} \wedge$
$\{\mathcal{V}^{\mathcal{A}}[\![t']\!] \mid t' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_t)\} = \{\mathcal{V}^{\mathcal{A}}[\![u']\!] \mid u' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, u_t)\} \wedge$ constr.prop.
$\{\mathcal{V}^{\mathcal{A}}[\![u']\!] \mid u' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, u_t)\} \subseteq \{\mathcal{V}^{\mathcal{A}}[\![u']\!] \mid u' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_u)\}$ $u_t \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_u)$
$\Rightarrow \mathcal{V}^{\mathcal{A}}[\![s]\!] \in \{\mathcal{V}^{\mathcal{A}}[\![u']\!] \mid u' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_u)\}$
$\Rightarrow$ there is $ct_u \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![u]\!])$ such that
$\mathcal{V}^{\mathcal{A}}[\![s]\!] \in \{\mathcal{V}^{\mathcal{A}}[\![u']\!] \mid u' \in T^{\leq}_{\mathtt{DObj}}(\emptyset, ct_u)\}$
$\Rightarrow \mathcal{A} \models s \in cont^{\mathcal{A}}_{\mathtt{DObj}}(u)$ Lemma 3.3.3(1)
$\Rightarrow (s,u) \in R^{cont}_{\mathcal{A}}$

(3) We conclude

$\mathcal{A} \models f(t_1, ..., t_n) = t \wedge f \notin \mathcal{C}_\tau$
$\Rightarrow \mathcal{A} \models f(ct_1, ..., ct_n) = t \wedge f \notin \mathcal{C}_\tau$
for $ct_i \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t_i]\!])(1 \leq i \leq n)$ — Defn. 3.2.9
$\Rightarrow$ there is $ct_t \in T^{\leq}_{\texttt{DObj}}(\emptyset, f(ct_1, ..., ct_n))$ such that $\mathcal{A} \models ct_t = t \wedge$ — Defn. 3.3.2
$f \notin \mathcal{C}_\tau \wedge ct_i \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t_i]\!])(1 \leq i \leq n)$ — constr.prop.(1)
$\Rightarrow$ there is $ct_t \in \bigcup_i T^{\leq}_{\texttt{DObj}}(\emptyset, ct_i)$ such that $\mathcal{A} \models ct_t = t \wedge$ — $f \notin \mathcal{C}_\tau$
$ct_i \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t_i]\!])(1 \leq i \leq n)$
$\Rightarrow$ there is $k \in \{1, ..., n\}, ct_t \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct_k)$ such that $\mathcal{A} \models ct_t = t \wedge$
$ct_k \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t_k]\!])$
$\Rightarrow$ there is $k \in \{1, ..., n\}$ such that $\mathcal{V}^{\mathcal{A}}[\![t]\!] = \mathcal{V}^{\mathcal{A}}[\![ct_t]\!]$ and
$\mathcal{V}^{\mathcal{A}}[\![ct_t]\!] \in \{\mathcal{V}^{\mathcal{A}}[\![t']\!] \mid t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct_k)\}$
$\Rightarrow$ there is $k \in \{1, ..., n\}$ such that $\mathcal{A} \models t \in cont^{\mathcal{A}}_{\texttt{DObj}}(t_k)$ — Lemma 3.3.3(1)

□

Proof C.1.6 (Proof of Lemma 3.3.4) We show the properties in consecutive order.

(1)
$\texttt{True} \Rightarrow \mathcal{A}' \models t = t$
$\Rightarrow \mathcal{A}' \models \neg(t \in \texttt{existDObj} \wedge t \neq t)$
$\Rightarrow \mathcal{A} \models t \notin \texttt{existDObj}$ — req. (3)

(2)
$\texttt{True} \Rightarrow \mathcal{A}' \models \texttt{oid}(t) = \texttt{oid}(t)$
$\Rightarrow \mathcal{A}' \models \neg(\texttt{oid}(t) \in \texttt{usedOIDs} \wedge \texttt{oid}(t) \neq \texttt{oid}(t))$
$\Rightarrow \mathcal{A} \models \texttt{oid}(t) \notin \texttt{usedOIDs}$ — req. (4)

The result for $tr \equiv \mathsf{del}(t)$ follows analogously. □

Proof C.1.7 (Proof of Lemma 3.3.5) We show that all three basic state changes preserve properties (1) to (4). We start with object creation.

$tr = \mathsf{cre}(t)$:

(1) Suppose $\mathcal{A} \models \forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow \texttt{oid}(x) \in \texttt{usedOIDs}$ and $(\mathcal{A}, \mathsf{cre}(t)) \overset{btr}{\rightsquigarrow} \mathcal{A}'$. We conclude
$\mathcal{A}' \models x \in \texttt{existDObj}$
$\Rightarrow \mathcal{A}' \models (x \in \texttt{existDObj} \wedge x = t) \vee (x \in \texttt{existDObj} \wedge x \neq t)$
$\Rightarrow \mathcal{A}' \models \texttt{oid}(x) \in \texttt{usedOIDs} \vee (x \in \texttt{existDObj} \wedge x \neq t)$ — req. (2)
$\Rightarrow \mathcal{A}' \models \texttt{oid}(x) \in \texttt{usedOIDs} \vee \mathcal{A} \models x \in \texttt{existDObj}$ — req. (3)
$\Rightarrow \mathcal{A}' \models \texttt{oid}(x) \in \texttt{usedOIDs} \vee \mathcal{A} \models \texttt{oid}(x) \in \texttt{usedOIDs}$ — prereq.
$\Rightarrow \mathcal{A}' \models \texttt{oid}(x) \in \texttt{usedOIDs} \vee (\texttt{oid}(x) \neq \texttt{oid}(t) \wedge \texttt{oid}(x) \in \texttt{usedOIDs})$ — req. (4)
$\Rightarrow \mathcal{A}' \models \texttt{oid}(x) \in \texttt{usedOIDs}$

(2) In the proof we will need the following property, which will be referred to by (*): For all $x : \texttt{DObj}$ it is true that $\mathcal{A} \models x \in \texttt{existDObj}$ implies $\mathcal{A}' \models \texttt{oid}(x) \neq \texttt{oid}(t)$. The proof goes es follows:
$\mathcal{A} \models x \in \texttt{existDObj} \Rightarrow \mathcal{A} \models \texttt{oid}(x) \in \texttt{usedOIDs}$ — prop. (1)
$\Rightarrow \mathcal{A}' \models \texttt{oid}(x) \in \texttt{usedOIDs} \wedge \texttt{oid}(x) \neq \texttt{oid}(t)$ — req. (4)
$\Rightarrow \mathcal{A}' \models \texttt{oid}(x) \neq \texttt{oid}(t)$

Now, we are ready to prove property (2) of Lemma 3.3.5. Suppose $\mathcal{A} \models \forall x, x' : \texttt{DObj} \bullet (x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj}) \Rightarrow (\texttt{oid}(x) = \texttt{oid}(x') \Rightarrow x = x')$ and $(\mathcal{A}, \mathsf{cre}(t)) \overset{btr}{\rightsquigarrow} \mathcal{A}'$. First, we conclude

$\mathcal{A}' \models x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj} \wedge \texttt{oid}(x) = \texttt{oid}(x')$
$\Rightarrow \mathcal{A}' \models x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj} \wedge \texttt{oid}(x) = \texttt{oid}(x') \wedge x = t \vee$
$\quad \mathcal{A}' \models x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj} \wedge \texttt{oid}(x) = \texttt{oid}(x') \wedge x \neq t$

Now, we derive $\mathcal{A}' \models x = x'$ for both sides of the disjunction.

$\mathcal{A}' \models x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj} \wedge \texttt{oid}(x) = \texttt{oid}(x') \wedge x = t$
$\Rightarrow \mathcal{A}' \models x' \in \texttt{existDObj} \wedge \texttt{oid}(x') = \texttt{oid}(t) \wedge x = t$
$\Rightarrow \mathcal{A} \models x' \notin \texttt{existDObj} \wedge \mathcal{A}' \models x' \in \texttt{existDObj} \wedge x = t$ (*)
$\Rightarrow \mathcal{A}' \models (x' \notin \texttt{existDObj} \vee x' = t) \wedge x' \in \texttt{existDObj} \wedge x = t$ req. (3)
$\Rightarrow \mathcal{A}' \models x' = t \wedge x = t$
$\Rightarrow \mathcal{A}' \models x' = x$

We remain to show the derivation for the other side of the disjunction.

$\mathcal{A}' \models x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj} \wedge \texttt{oid}(x) = \texttt{oid}(x') \wedge x \neq t$
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge \mathcal{A}' \models x' \in \texttt{existDObj} \wedge \texttt{oid}(x) = \texttt{oid}(x')$ req. (3)
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge$
$\quad \mathcal{A}' \models x' \in \texttt{existDObj} \wedge \texttt{oid}(x) = \texttt{oid}(x') \wedge \texttt{oid}(x) \neq \texttt{oid}(t)$ (*)
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge \mathcal{A}' \models x' \in \texttt{existDObj} \wedge \texttt{oid}(x') \neq \texttt{oid}(t)$
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge \mathcal{A}' \models x' \in \texttt{existDObj} \wedge x' \neq t$
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj}$ req. (3)
$\Rightarrow \mathcal{A} \models x = x'$ prereq.
$\Rightarrow \mathcal{A}' \models x = x'$ req. (5)

(3) Suppose $\mathcal{A} \models \forall x, x' : \texttt{DObj} \bullet (x \in \texttt{existDObj} \wedge x' \in \mathit{cont}^{\mathcal{A}}_{\texttt{DObj}}(x)) \Rightarrow x' \in \texttt{existDObj}$ and $(\mathcal{A}, \mathsf{cre}(t)) \overset{btr}{\rightsquigarrow} \mathcal{A}'$. First, we conclude

$\mathcal{A}' \models x \in \texttt{existDObj} \wedge x' \in \mathit{cont}^{\mathcal{A}'}_{\texttt{DObj}}(x)$
$\Rightarrow \mathcal{A}' \models x \in \texttt{existDObj} \wedge x' \in \mathit{cont}^{\mathcal{A}'}_{\texttt{DObj}}(x) \wedge x \neq t \vee$
$\quad \mathcal{A}' \models x \in \texttt{existDObj} \wedge x' \in \mathit{cont}^{\mathcal{A}'}_{\texttt{DObj}}(x) \wedge x = t$

Now, we derive $\mathcal{A}' \models x' \in \texttt{existDObj}$ for both sides of the disjunction.

$\mathcal{A}' \models x \in \texttt{existDObj} \wedge x' \in \mathit{cont}^{\mathcal{A}'}_{\texttt{DObj}}(x) \wedge x \neq t$
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge \mathcal{A}' \models x' \in \mathit{cont}^{\mathcal{A}'}_{\texttt{DObj}}(x)$ req. (3)
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge x' \in \mathit{cont}^{\mathcal{A}}_{\texttt{DObj}}(x)$ req. (5)
$\Rightarrow \mathcal{A} \models x' \in \texttt{existDObj}$ prereq.
$\Rightarrow \mathcal{A}' \models x' \in \texttt{existDObj} \wedge x' \neq t$ req. (3)
$\Rightarrow \mathcal{A}' \models x' \in \texttt{existDObj}$

We remain to show the derivation for the other side of the disjunction.

$\mathcal{A}' \models x \in \texttt{existDObj} \wedge x' \in \mathit{cont}^{\mathcal{A}'}_{\texttt{DObj}}(x) \wedge x = t$
$\Rightarrow \mathcal{A}' \models x' \in \mathit{cont}^{\mathcal{A}'}_{\texttt{DObj}}(t)$
$\Rightarrow \mathcal{A}' \models x' = t \vee \mathcal{A}' \models x' \neq t \wedge x' \in \mathit{cont}^{\mathcal{A}'}_{\texttt{DObj}}(t)$
$\Rightarrow \mathcal{A}' \models x' \in \texttt{existDObj} \vee \mathcal{A}' \models x' \neq t \wedge x' \in \mathit{cont}^{\mathcal{A}'}_{\texttt{DObj}}(t)$ req. (2)
$\Rightarrow \mathcal{A}' \models x' \in \texttt{existDObj} \vee \mathcal{A} \models x' \neq t \wedge x' \in \mathit{cont}^{\mathcal{A}}_{\texttt{DObj}}(t)$ req. (5)
$\Rightarrow \mathcal{A}' \models x' \in \texttt{existDObj} \vee$
$\quad (\mathcal{V}^{\mathcal{A}}[\![x']\!] \neq \mathcal{V}^{\mathcal{A}}[\![t]\!] \wedge$ Defn. 3.2.11
$\quad$ there is $ct_t \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!])$ such that
$\quad \mathcal{V}^{\mathcal{A}}[\![x']\!] \in \{\mathcal{V}^{\mathcal{A}}[\![t']\!] \mid t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct_t)\})$ Lemma 3.3.3(1)
$\Rightarrow \mathcal{A}' \models x' \in \texttt{existDObj} \vee$
$\quad (\mathcal{V}^{\mathcal{A}}[\![x']\!] \neq \mathcal{V}^{\mathcal{A}}[\![t]\!] \wedge \mathcal{V}^{\mathcal{A}}[\![x']\!] \in \{\mathcal{V}^{\mathcal{A}}[\![t']\!] \mid t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, t)\})$ constr.prop.(2)
$\quad$ t min.constr.term
$\Rightarrow \mathcal{A}' \models x' \in \texttt{existDObj} \vee$
$\quad \mathcal{V}^{\mathcal{A}}[\![x']\!] \in \{\mathcal{V}^{\mathcal{A}}[\![t']\!] \mid t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, t) \backslash \{t\}\}$
$\Rightarrow \mathcal{A}' \models x' \in \texttt{existDObj} \vee \mathcal{A} \models x' \in \texttt{existDObj}$ req. (1), Defn. 3.2.11
$\Rightarrow \mathcal{A}' \models x' \in \texttt{existDObj} \vee (x' \in \texttt{existDObj} \wedge x' \neq t)$ req. (3)
$\Rightarrow \mathcal{A}' \models x' \in \texttt{existDObj}$

(4) Given an appropriate t_{id}, we conclude

$\mathcal{A} \models t_{id} \in \texttt{usedOIDs} \wedge \forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow \texttt{oid}(x) \neq t_{id}$
$\Rightarrow \mathcal{A}' \models t_{id} \in \texttt{usedOIDs} \wedge \texttt{oid}(t) \neq t_{id} \wedge$ req. (4)
$\mathcal{A} \models t_{id} \in \texttt{usedOIDs} \wedge \forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow \texttt{oid}(x) \neq t_{id}$
$\Rightarrow \mathcal{A}' \models t_{id} \in \texttt{usedOIDs} \wedge \texttt{oid}(t) \neq t_{id} \wedge$
$\mathcal{A}' \models \forall x : \texttt{DObj} \bullet (x \in \texttt{existDObj} \wedge x \neq t) \Rightarrow \texttt{oid}(x) \neq t_{id}$ req. (3),(4)
$\Rightarrow \mathcal{A}' \models t_{id} \in \texttt{usedOIDs} \wedge \forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow \texttt{oid}(x) \neq t_{id}$

$tr = \mathsf{trans}(t_{src} \mapsto t_{trg})$:
Since transformation uses object creation it is easy to see that the properties hold.

$tr = \mathsf{del}(t)$:

(1) Suppose $\mathcal{A} \models \forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow \texttt{oid}(x) \in \texttt{usedOIDs}$ and $(\mathcal{A}, \mathsf{del}(t)) \overset{btr}{\rightsquigarrow} \mathcal{A}'$. We conclude
$\mathcal{A}' \models x \in \texttt{existDObj} \Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge x \neq t$ req. (3)
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj}$
$\Rightarrow \mathcal{A} \models \texttt{oid}(x) \in \texttt{usedOIDs}$ prereq.
$\Rightarrow \mathcal{A}' \models \texttt{oid}(x) \in \texttt{usedOIDs}$ req. (4)

(2) Suppose $\mathcal{A} \models \forall x, x' : \texttt{DObj} \bullet (x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj}) \Rightarrow (\texttt{oid}(x) = \texttt{oid}(x') \Rightarrow x = x')$ and $(\mathcal{A}, \mathsf{del}(t)) \overset{btr}{\rightsquigarrow} \mathcal{A}'$. We conclude
$\mathcal{A}' \models x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj} \wedge \texttt{oid}(x) = \texttt{oid}(x')$
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge x \neq t \wedge x' \in \texttt{existDObj} \wedge x' \neq t \wedge$ req. (3)
$\mathcal{A}' \models \texttt{oid}(x) = \texttt{oid}(x')$
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj} \wedge \mathcal{A}' \models \texttt{oid}(x) = \texttt{oid}(x')$
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge x' \in \texttt{existDObj} \wedge \texttt{oid}(x) = \texttt{oid}(x')$ req. (5)
$\Rightarrow \mathcal{A} \models x = x'$ prereq.
$\Rightarrow \mathcal{A}' \models x = x'$ req. (5)

(3) Suppose $\mathcal{A} \models \forall x, x' : \texttt{DObj} \bullet (x \in \texttt{existDObj} \wedge x' \in \mathit{cont}^{\mathcal{A}}_{\texttt{DObj}}(x)) \Rightarrow x' \in \texttt{existDObj}$ and $(\mathcal{A}, \mathsf{cre}(t)) \overset{btr}{\rightsquigarrow} \mathcal{A}'$. Then we conclude
$\mathcal{A}' \models x \in \texttt{existDObj} \wedge x' \in \mathit{cont}^{\mathcal{A}'}_{\texttt{DObj}}(x)$
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge x \neq t \wedge \mathcal{A}' \models x' \in \mathit{cont}^{\mathcal{A}'}_{\texttt{DObj}}(x)$ req. (3)
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge x \neq t \wedge x' \in \mathit{cont}^{\mathcal{A}}_{\texttt{DObj}}(x)$ req. (5)
$\Rightarrow \mathcal{A} \models x \in \texttt{existDObj} \wedge x' \in \mathit{cont}^{\mathcal{A}}_{\texttt{DObj}}(x) \wedge x' \in \texttt{existDObj} \wedge x \neq t$ prereq.
$\Rightarrow \mathcal{A} \models x' \in \texttt{existDObj} \wedge x \neq t \wedge (x = t \vee x' \neq t)$ req. (1)
$\Rightarrow \mathcal{A} \models x' \in \texttt{existDObj} \wedge x' \neq t$
$\Rightarrow \mathcal{A}' \models x' \in \texttt{existDObj}$ req. (3)

(4) Given an appropriate t_{id}, we conclude
$\mathcal{A} \models t_{id} \in \texttt{usedOIDs} \wedge \forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow \texttt{oid}(x) \neq t_{id}$
$\Rightarrow \mathcal{A}' \models t_{id} \in \texttt{usedOIDs} \wedge$ req. (4)
$\mathcal{A}' \models \forall x : \texttt{DObj} \bullet (x \in \texttt{existDObj} \vee x = t) \Rightarrow \texttt{oid}(x) \neq t_{id}$ req. (3)
$\Rightarrow \mathcal{A}' \models t_{id} \in \texttt{usedOIDs} \wedge \forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow \texttt{oid}(x) \neq t_{id}$

□

C.2 Proofs for Chap. 4

Proof C.2.1 (Proof of Lemma 4.3.1) We show the property by induction on the structure of t.

Case 1 ($t \equiv x$): By definition $\mathcal{V}^{\mathcal{A}}_d[\![x]\!]\eta = \eta(x) = \mathcal{V}^{\mathcal{A}}[\![x]\!]\eta$.

Case 2 ($t \equiv f(t_1, ..., t_n)$): We distinguish two cases.

Case 2.1 (τ<DObj): We conclude

$\bot \neq \mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta \Rightarrow \mathcal{A} \models t \in \texttt{existDObj}[\eta]$ Tab. 4.4
$\Rightarrow \mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta$ Tab. 4.4

Case 2.2 ($\neg\tau$<DObj): We conclude

$\bot \neq \mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta \Rightarrow \exists c(t'_1,...,t'_m) \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \bullet$
$\bot \neq \mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = c^{\mathcal{A}^\bot}(\mathcal{V}_d^{\mathcal{A}}[\![t'_1]\!]\eta,...,\mathcal{V}_d^{\mathcal{A}}[\![t'_m]\!]\eta)$ Tab. 4.4
$\Rightarrow c(t'_1,...,t'_m) \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \wedge$
$\mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = c^{\mathcal{A}^\bot}(\mathcal{V}_d^{\mathcal{A}}[\![t'_1]\!]\eta,...,\mathcal{V}_d^{\mathcal{A}}[\![t'_m]\!]\eta) \wedge$
$\forall i \in \{1,...,m\} \bullet \mathcal{V}_d^{\mathcal{A}}[\![t'_i]\!]\eta \neq \bot$ Defn. 3.2.3
$\Rightarrow c(t'_1,...,t'_m) \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \wedge$
$\mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = c^{\mathcal{A}^\bot}(\mathcal{V}_d^{\mathcal{A}}[\![t'_1]\!]\eta,...,\mathcal{V}_d^{\mathcal{A}}[\![t'_m]\!]\eta) \wedge$
$\forall i \in \{1,...,m\} \bullet \mathcal{V}_d^{\mathcal{A}}[\![t'_i]\!]\eta \neq \bot \wedge \mathcal{V}_d^{\mathcal{A}}[\![t'_i]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![t'_i]\!]\eta$ ind.hyp.
$\Rightarrow c(t'_1,...,t'_m) \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \wedge$
$\mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = c^{\mathcal{A}^\bot}(\mathcal{V}^{\mathcal{A}}[\![t'_1]\!]\eta,...,\mathcal{V}^{\mathcal{A}}[\![t'_m]\!]\eta)$
$\Rightarrow c(t'_1,...,t'_m) \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \wedge$
$\mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![c(t'_1,...,t'_m)]\!]\eta$ Tab. 3.2
$\Rightarrow \mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta$ Defn. 3.2.9

□

Proof C.2.2 (Proof of Thm. 4.3.1) We prove the properties in consecutive order.

(1) We show both directions by induction on the structure of $t \in T_\tau(X, \Sigma)$.

Case 1 ($t \equiv x$):

($\Longrightarrow$)

We conclude:

$\bot \neq \mathcal{V}^{\mathcal{A}}[\![x]\!]\eta = \mathcal{V}_d^{\mathcal{A}}[\![x]\!]\eta$
$\Rightarrow \eta(x) \neq \bot \wedge \eta \in \tau^{\mathcal{A}}$ Defn. 3.2.4
$\Rightarrow (\exists ct \in CT_{\mathcal{A}}^{min}(\Sigma, \eta(x)) \bullet \mathcal{V}_d^{\mathcal{A}}[\![ct]\!] = \eta(x)) \wedge \eta(x) \neq \bot$ prereq. in Thm. 4.3.1
$\Rightarrow \mathcal{V}^{\mathcal{A}}[\![x]\!]\eta = \mathcal{V}_d^{\mathcal{A}}[\![ct]\!] \wedge \mathcal{V}_d^{\mathcal{A}}[\![ct]\!] \neq \bot$ Defn. 3.2.6
$\Rightarrow \mathcal{V}^{\mathcal{A}}[\![x]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![ct]\!] \wedge$ Lemma 4.3.1
$\forall y : \texttt{DObj} \bullet y \in cont_{\texttt{DObj}}^{\mathcal{A}}(ct) \Rightarrow y \in \texttt{existDObj}$ ind.hyp.
$\Rightarrow \forall y : \texttt{DObj} \bullet y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(x) \Rightarrow y \in \texttt{existDObj}$ Defn. 3.3.3

($\Longleftarrow$)

By definition $\mathcal{V}_d^{\mathcal{A}}[\![x]\!]\eta = \eta(x) = \mathcal{V}^{\mathcal{A}}[\![x]\!]\eta$.

Case 2 ($t \equiv f(t_1,...t_n)$):

($\Longrightarrow$)

Assume $\bot \neq \mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta$. We have to distinguish two cases.

Case 1 (τ<DObj): According to Tab. 4.4 this implies $\mathcal{A} \models t \in \texttt{existDObj}[\eta]$. If τ<DObj $\wedge\, \mathcal{A} \not\models t \in \texttt{existDObj}[\eta]$ was true, $\mathcal{V}_d^{\mathcal{A}}[\![x]\!]\eta = \bot$ would hold. Therefore, we conclude:

$t \in \texttt{existDObj} \wedge$ $\mathcal{A}$ derived by basic transitions
$\Rightarrow t \in \texttt{existDObj} \wedge$
$\mathcal{A} \models \forall x, x' : \texttt{DObj} \bullet$
$x \in \texttt{existDObj} \wedge x' \in cont_{\texttt{DObj}}^{\mathcal{A}}(x) \Rightarrow x' \in \texttt{existDObj}[\eta]$ Cor. 3.3.1(3)
$\Rightarrow \mathcal{A} \models \forall y : \texttt{DObj} \bullet y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(t) \Rightarrow y \in \texttt{existDObj}[\eta]$

Case 2 ($\neg\tau$<DObj): We conclude

$\bot \neq \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta = \mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta$
$\Rightarrow \exists c(t'_1,...,t'_m) \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \bullet$
$\quad \bot \neq \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta = c^{\mathcal{A}^\bot}(\mathcal{V}_d^{\mathcal{A}}[\![t'_1]\!]\eta, ..., \mathcal{V}_d^{\mathcal{A}}[\![t'_m]\!]\eta)$ Tab. 4.4
$\Rightarrow c(t'_1,...,t'_m) \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \wedge$
$\quad \forall i \in \{1,...,m\} \bullet \mathcal{V}_d^{\mathcal{A}}[\![t'_i]\!]\eta \neq \bot$ Defn. 3.2.3
$\Rightarrow c(t'_1,...,t'_m) \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \wedge$
$\quad \forall i \in \{1,...,m\} \bullet \mathcal{V}_d^{\mathcal{A}}[\![t'_i]\!]\eta \neq \bot \wedge \mathcal{V}_d^{\mathcal{A}}[\![t'_i]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![t'_i]\!]\eta$ Lemma 4.3.1
$\Rightarrow c(t'_1,...,t'_m) \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \wedge$
$\quad \forall i \in \{1,...,m\} \bullet \mathcal{A} \models \forall y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(t'_i) \bullet y \in \texttt{existDObj}[\eta]$ ind.hyp.
$\Rightarrow c(t'_1,...,t'_m) \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \wedge$ $\neg\tau$<DObj,
$\quad \mathcal{A} \models \forall y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(c(t'_1,...,t'_m)) \bullet y \in \texttt{existDObj}[\eta]$ Defn. 3.3.3
$\Rightarrow \mathcal{A} \models \forall y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(t) \bullet y \in \texttt{existDObj}[\eta]$ Lemma 3.3.3(1)

($\Longleftarrow$)

We distinguish τ<DObj and $\neg\tau$<DObj. We shall see that τ<DObj implies $t \in$ existDObj such that the case τ<DObj, $\mathcal{A} \not\models t \in \texttt{existDObj}[\eta]$ in Tab. 4.4 can be neglected.

Case 1 (τ<DObj): We conclude:

$\mathcal{A} \models t \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(t)[\eta] \wedge$ $\mathcal{A}$ derived by basic transitions — Defn. 3.3.3, prereq. of Thm. 4.3.1
$\Rightarrow \mathcal{A} \models t \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(t)[\eta] \wedge$
$\quad \mathcal{A} \models \forall x, x' : \texttt{DObj} \bullet$
$\quad\quad x \in \texttt{existDObj} \wedge x' \in cont_{\texttt{DObj}}^{\mathcal{A}}(x) \Rightarrow x' \in \texttt{existDObj}[\eta]$ Cor. 3.3.1(3)
$\Rightarrow \mathcal{A} \models t \in \texttt{existDObj}[\eta]$
$\Rightarrow \mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta$ Tab. 4.4

Case 2 ($\neg\tau$<DObj): By induction on the structure of t we conclude:

$\mathcal{A} \models \forall y : \texttt{DObj} \bullet y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(t) \Rightarrow y \in \texttt{existDObj}[\eta] \wedge$ prereq. of
$\mathcal{V}^{\mathcal{A}}[\![t]\!]\eta \neq \bot$ Thm. 4.3.1
$\Rightarrow \forall ct \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \bullet$
$\quad \mathcal{A} \models \forall y : \texttt{DObj} \bullet y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(ct) \Rightarrow y \in \texttt{existDObj}[\eta] \wedge$ Defn. 3.3.3
$\quad \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta \neq \bot$
$\Rightarrow \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta \neq \bot \wedge$
$\quad \forall ct \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \bullet (\mathcal{V}^{\mathcal{A}}[\![t]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![ct]\!]\eta \wedge$
$\quad\quad \mathcal{A} \models \forall y : \texttt{DObj} \bullet y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(ct) \Rightarrow y \in \texttt{existDObj}[\eta])$ Defn. 3.2.9
$\Rightarrow \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta \neq \bot \wedge$
$\quad \forall ct \in CT_{\mathcal{A}}^{min}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \bullet$ ind.hyp.
$\quad\quad \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![ct]\!]\eta \wedge \mathcal{V}^{\mathcal{A}}[\![ct]\!]\eta = \mathcal{V}_d^{\mathcal{A}}[\![ct]\!]\eta$ ($l(ct) < l(t)$ as $\neg\tau$<DObj)
$\Rightarrow \mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta$ Tab. 4.4

(2) Both directions are direct results of property (1) and Lemma 4.3.1. We conclude:

$\mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = \bot \Leftrightarrow \mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta = \bot \wedge \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta \neq \bot$ prereq. of Thm. 4.3.1
$\Leftrightarrow \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta \neq \bot \wedge \mathcal{V}_d^{\mathcal{A}}[\![t]\!]\eta \neq \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta$ Lemma 4.3.1(for $\Leftarrow$)
$\Leftrightarrow \mathcal{A} \models \neg(\forall y : \texttt{DObj} \bullet y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(t) \Rightarrow y \in \texttt{existDObj})[\eta]$ Thm. 4.3.1(1)
$\Leftrightarrow \mathcal{A} \models \exists y : \texttt{DObj} \bullet \neg(y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(t) \Rightarrow y \in \texttt{existDObj})[\eta]$
$\Leftrightarrow \mathcal{A} \models \exists y : \texttt{DObj} \bullet y \in cont_{\texttt{DObj}}^{\mathcal{A},\eta}(t) \wedge y \notin \texttt{existDObj})[\eta]$

□

Proof C.2.3 (Proof of Lemma 4.3.2) We prove the assumptions consecutively.

(1) We conclude:

$dom(\mathcal{A}, x : \tau, \eta) = (\mathcal{A}, d) \wedge a \in d$
$\Rightarrow \eta \in Env(X, \mathcal{A}) \wedge a \in \{v \mid v \in \tau^{\mathcal{A}}, ct \in CT^{min}_{\mathcal{A}}(\Sigma, v), (\mathcal{A}, ct) \overset{t_d}{\rightsquigarrow} v\}$ Defn. 4.3.2
$\Rightarrow \eta \in Env(X, \mathcal{A}) \wedge a \in \tau^{\mathcal{A}}$

(2) We conclude:

$a \in d \Leftrightarrow \exists v \in \tau^{\mathcal{A}}, ct \in CT^{min}_{\mathcal{A}}(\Sigma, v) \bullet a = v \wedge (\mathcal{A}, ct) \overset{t_d}{\rightsquigarrow} v$ Defn. 4.3.2
$\Leftrightarrow \exists v \in \tau^{\mathcal{A}}, ct \in CT^{min}_{\mathcal{A}}(\Sigma, v) \bullet$
$\quad a = v \wedge \mathcal{A} \models \forall y : \texttt{DObj} \bullet y \in cont^{\mathcal{A}}_{\texttt{DObj}}(ct) \Rightarrow y \in \texttt{existDObj}$ Thm. 4.3.1(1)
$\Leftrightarrow \exists v \in \tau^{\mathcal{A}}, ct \in CT^{min}_{\mathcal{A}}(\Sigma, v) \bullet$ Cor. 3.3.1(3) (for $\Leftarrow$)
$\quad a = v \wedge \mathcal{A} \models ct \in \texttt{existDObj}$ Defn. 3.3.3 (for $\Rightarrow$)
$\Leftrightarrow \exists v \in \tau^{\mathcal{A}}, t \in GT_\tau(\Sigma) \bullet$ Defn. 3.2.9(for $\Rightarrow$)
$\quad a = v \wedge \mathcal{V}^{\mathcal{A}}_d[\![t]\!] = v \wedge \mathcal{A} \models t \in \texttt{existDObj}$ Defn. 3.2.12(for $\Leftarrow$)
$\Leftrightarrow a \in \tau^{\mathcal{A}} \cap \{\mathcal{V}^{\mathcal{A}}_d[\![t]\!] \mid t \in GT_\tau(\Sigma), \mathcal{A} \models t \in \texttt{existDObj}\}$

□

Proof C.2.4 (Proof of Thm. 4.3.2)

(1) We prove the assumption by induction on the structure of ϕ. Throughout the proof we use (*) when referring to the prerequisite that the object-valued content of all terms in T_ϕ exists.

Case 1 ($\phi \equiv s = t$): Then $T_\phi = \{s, t\}$ and we conclude:

$\mathcal{A} \models s = t[\eta] \Leftrightarrow \bot \neq \mathcal{V}^{\mathcal{A}}[\![s]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta$ Tab. 3.3
$\Leftrightarrow \bot \neq \mathcal{V}^{\mathcal{A}}[\![s]\!]\eta = \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta \wedge$ (*),
$\quad \mathcal{V}^{\mathcal{A}}[\![s]\!]\eta = \mathcal{V}^{\mathcal{A}}_d[\![s]\!]\eta \wedge \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta = \mathcal{V}^{\mathcal{A}}_d[\![t]\!]\eta \wedge$ Thm. 4.3.1(1)
$\quad \bot \neq \mathcal{V}^{\mathcal{A}}_d[\![s]\!]\eta = \mathcal{V}^{\mathcal{A}}_d[\![t]\!]\eta$
$\Leftrightarrow \mathcal{A} \models_d s = t[\eta]$ Tab. 3.3,Tab. 4.4

Case 2 ($\phi \equiv p(t_1, ..., t_n)$): Then $T_\phi = \{t_1, ..., t_n\}$ and we conclude

$\mathcal{A} \models p(t_1, ..., t_n)[\eta]$
$\Leftrightarrow \exists p_{\overline{\tau_i}} \in \mathcal{P} \bullet (\mathcal{V}^{\mathcal{A}}[\![t_1]\!]\eta, ..., \mathcal{V}^{\mathcal{A}}[\![t_n]\!]\eta) \in p^{\mathcal{A}}_{\overline{\tau_i}}$ Tab. 3.3
$\Leftrightarrow \exists p_{\overline{\tau_i}} \in \mathcal{P} \bullet (\mathcal{V}^{\mathcal{A}}[\![t_1]\!]\eta, ..., \mathcal{V}^{\mathcal{A}}[\![t_n]\!]\eta) \in p^{\mathcal{A}}_{\overline{\tau_i}} \wedge$
$\quad \forall i \in \{1, ..., n\} \bullet \mathcal{V}^{\mathcal{A}}[\![t_i]\!]\eta \neq \bot$ Defn. 3.2.3
$\Leftrightarrow \exists p_{\overline{\tau_i}} \in \mathcal{P} \bullet (\mathcal{V}^{\mathcal{A}}_d[\![t_1]\!]\eta, ..., \mathcal{V}^{\mathcal{A}}_d[\![t_n]\!]\eta) \in p^{\mathcal{A}}_{\overline{\tau_i}} \wedge$ (*),
$\quad \forall i \in \{1, ..., n\} \bullet \mathcal{V}^{\mathcal{A}}_d[\![t_i]\!]\eta \neq \bot$ Thm. 4.3.1(1)
$\Leftrightarrow \mathcal{A} \models_d p(t_1, ..., t_n)[\eta]$ Tab. 3.3,Tab. 4.4

Case 3 ($\phi \equiv \psi \wedge \psi'$): The assumption holds by induction hypothesis.

Case 4 ($\phi \equiv \neg\psi$): The assumption holds by induction hypothesis.

(2) We conclude:

$\mathcal{A} \models \forall x : \tau \bullet \phi[\eta]$
$\Rightarrow \mathcal{A} \models \phi[\eta[x \mapsto a]]$ for all $a \in \tau^{\mathcal{A}}$ Tab. 3.3
$\Rightarrow \mathcal{A} \models \phi[\eta[x \mapsto a]]$ for all
$\quad a \in \{v \mid v \in \tau^{\mathcal{A}}, \mathcal{A} \models \forall y : \texttt{DObj} \bullet y \in cont^{\mathcal{A}, \eta[x \mapsto v]}_{\texttt{DObj}}(x)[\eta[x \mapsto v]]\}$
$\Rightarrow \mathcal{A} \models \phi[\eta[x \mapsto a]]$ for all
$\quad a \in \{v \mid v \in \tau^{\mathcal{A}}, ct \in CT^{min}_{\mathcal{A}}(\Sigma, \mathcal{V}^{\mathcal{A}}[\![x]\!]\eta[x \mapsto v]),$
$\quad\quad \mathcal{A} \models \forall y : \texttt{DObj} \bullet y \in cont^{\mathcal{A}}_{\texttt{DObj}}(ct)\}$ Defn. 3.2.9
$\Rightarrow \mathcal{A} \models \phi[\eta[x \mapsto a]]$ for all
$\quad a \in \{v \mid v \in \tau^{\mathcal{A}}, ct \in CT^{min}_{\mathcal{A}}(\Sigma, v), \mathcal{A} \models \forall y : \texttt{DObj} \bullet y \in cont^{\mathcal{A}}_{\texttt{DObj}}(ct)\}$ Tab. 3.2
$\Rightarrow \mathcal{A} \models \phi[\eta[x \mapsto a]]$ for all
$\quad a \in \{v \mid v \in \tau^{\mathcal{A}}, ct \in CT^{min}_{\mathcal{A}}(\Sigma, v), \mathcal{V}^{\mathcal{A}}_d[\![ct]\!] = v\}$ Thm. 4.3.1(1)
$\Rightarrow \mathcal{A} \models_d \forall x : \tau \bullet \phi[\eta]$ Defn. 4.3.2,
Tab. 3.3, (*),
Thm. 4.3.2(1)

(3) We conclude:

$\mathcal{A} \models \forall x : \tau \bullet x \in \texttt{existDObj} \Rightarrow \phi[\eta]$
$\Leftrightarrow \mathcal{A} \models x \in \texttt{existDObj} \Rightarrow \phi[\eta[x \mapsto a]]$ for all $a \in \tau^{\mathcal{A}}$ — Tab. 3.3
$\Leftrightarrow \mathcal{A} \models x \in \texttt{existDObj}[\eta[x \mapsto a]]$ implies $\mathcal{A} \models \phi[\eta[x \mapsto a]]$ for all $a \in \tau^{\mathcal{A}}$
$\Leftrightarrow \mathcal{A} \models \phi[\eta[x \mapsto a]]$ for all — Tab. 3.2,
$a \in \tau^{\mathcal{A}} \cap \{\mathcal{V}^{\mathcal{A}}[\![t]\!] \mid t \in GT_\tau(\Sigma), \mathcal{A} \models t \in \texttt{existDObj}\}$ — Tab. 3.3
$\Leftrightarrow \mathcal{A} \models \phi[\eta[x \mapsto a]]$ for all — Thm. 4.3.1(1)
$a \in \tau^{\mathcal{A}} \cap \{\mathcal{V}_d^{\mathcal{A}}[\![t]\!] \mid t \in GT_\tau(\Sigma), \mathcal{A} \models t \in \texttt{existDObj}\}$
$\Leftrightarrow \mathcal{A} \models \phi[\eta[x \mapsto a]]$ for all — Lemma 4.3.2(2)
$a \in \{v \mid v \in \tau^{\mathcal{A}}, ct \in CT_{\mathcal{A}}^{min}(\Sigma, v), \mathcal{V}_d^{\mathcal{A}}[\![ct]\!] = v\}$
$\Leftrightarrow \mathcal{A} \models_d \forall x : \tau \bullet \phi[\eta]$ — Defn. 4.3.2, Tab. 3.3, (*), Thm. 4.3.2(1)

□

Proof C.2.5 (Proof of Lemma 4.3.3) We prove the assumptions in consecutive order.

(1) This directly follows from rule **Subtyping** in Tab. 4.5.

(2) We consider wildcard-free concept terms only. According to Tab. 4.6 the results apply to concept terms with wildcards in a straightforward way.
We have to show that $\eta[x_1 \mapsto v_{l(x_1)}]...[x_k \mapsto v_{l(x_k)}]$ is a suitable variable assignment in $Env(X, \mathcal{A})$ for ι_C and that v is in $\tau^{\mathcal{A}} \cup \{\bot\}$:

$\mathcal{K}(\overline{t_i})[C] \in KT_\tau(X, \Sigma, KD) \wedge (\mathcal{A}, \eta, \mathcal{K}(\overline{t_i})[C]) \overset{t_d}{\rightsquigarrow} v$
$\Rightarrow \exists \overline{\tau_i} \bullet \overline{t_i} \in \overline{T_{\tau_i}(X, \Sigma)} \wedge$
$\mathcal{K}\ \mathcal{I}\ \{C_1 = \iota_{C_1}, ..., C_m = \iota_{C_m}\} \in KD\} \wedge$
$C = \iota_C \in \{C_1 = \iota_{C_1}, ..., C_m = \iota_{C_m}\} \wedge \exists X' \bullet \iota_C \in T_\tau(X', \Sigma) \wedge$
$\mathcal{FV}(\iota_C) = \{x_1 : \tau_1', ..., x_k : \tau_k'\} \wedge$
$l : \mathcal{FV}(\iota_C) \to \{1, ..., n\} \wedge$
$\forall j \in \{1, ..., k\} \bullet ra(C, \mathcal{I})(x_j) = role(l(x_j), \mathcal{I}) \wedge$ — Tab. 4.6
$\forall x : \tau' \in \mathcal{FV}(\iota_C) \bullet ra(C, \mathcal{I})(x) = role(i, \mathcal{I}) \Rightarrow \tau_i < \tau' \wedge$ — Tab. 4.5
$v = \mathcal{V}_d^{\mathcal{A}}[\![\iota_C]\!]\eta[x_1 \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_1)}]\!]\eta]...[x_k \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_k)}]\!]\eta]$
$\Rightarrow \overline{t_i} \in \overline{T_{\tau_i}(X, \Sigma)} \ \wedge \ \iota_C \in T_\tau(X', \Sigma) \wedge$
$\mathcal{FV}(\iota_C) = \{x_1 : \tau_1', ..., x_k : \tau_k'\} \wedge$
$l : \mathcal{FV}(\iota_C) \to \{1, ..., n\} \wedge$
$\forall j \in \{1, ..., k\} \bullet \tau_{l(x_j)} < \tau_j' \wedge$
$v = \mathcal{V}_d^{\mathcal{A}}[\![\iota_C]\!]\eta[x_1 \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_1)}]\!]\eta]...[x_k \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_k)}]\!]\eta]$
$\Rightarrow \iota_C \in T_\tau(X', \Sigma) \wedge$
$\mathcal{FV}(\iota_C) = \{x_1 : \tau_1', ..., x_k : \tau_k'\} \wedge$
$l : \mathcal{FV}(\iota_C) \to \{1, ..., n\} \wedge$
$\forall j \in \{1, ..., k\} \bullet \tau_{l(x_j)} < \tau_j' \wedge \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_j)}]\!]\eta \in \tau_{l(x_j)}^{\mathcal{A}} \cup \{\bot\} \wedge$ — Cor. 4.3.1
$v = \mathcal{V}_d^{\mathcal{A}}[\![\iota_C]\!]\eta[x_1 \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_1)}]\!]\eta]...[x_k \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_k)}]\!]\eta]$
$\Rightarrow \iota_C \in T_\tau(X', \Sigma) \wedge$
$\mathcal{FV}(\iota_C) = \{x_1 : \tau_1', ..., x_k : \tau_k'\} \wedge$
$l : \mathcal{FV}(\iota_C) \to \{1, ..., n\} \wedge$
$\forall j \in \{1, ..., k\} \bullet \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_j)}]\!]\eta \in (\tau_j')^{\mathcal{A}} \cup \{\bot\} \wedge$
$v = \mathcal{V}_d^{\mathcal{A}}[\![\iota_C]\!]\eta[x_1 \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_1)}]\!]\eta]...[x_k \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_k)}]\!]\eta]$
$\Rightarrow \eta[x_1 \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_1)}]\!]\eta]...[x_k \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_k)}]\!]\eta] \in Env(X, \mathcal{A}) \wedge$ — Defn. 3.2.4
$\mathcal{V}_d^{\mathcal{A}}[\![\iota_C]\!]\eta[x_1 \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_1)}]\!]\eta]...[x_k \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_k)}]\!]\eta] \in \tau^{\mathcal{A}} \cup \{\bot\} \wedge$ — Cor. 4.3.1
$v = \mathcal{V}_d^{\mathcal{A}}[\![\iota_C]\!]\eta[x_1 \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_1)}]\!]\eta]...[x_k \mapsto \mathcal{V}_d^{\mathcal{A}}[\![t_{l(x_k)}]\!]\eta]$
$\Rightarrow v \in \tau^{\mathcal{A}} \cup \{\bot\}$ — Tab. 4.6

(3) In the proof of part (2) of this lemma role assignments (thus, the index labeling l) are uniquely determined. Also, we have used the value *function* $\mathcal{V}_d[\![]\!]$ in order to emphasize that the value for ι_C is unique. The property, thus, directly follows.
In presence of a wildcard _, in general, multiple values can be derived due to the existential quantification over all existing contexts in the concept semantics (cf. Tab. 4.6).

□

C.3 Proofs for Chap. 5

Proof C.3.1 (Proof of Lemma 5.2.1) Assume $\mathcal{K}$ is non-functional. Then we conclude:

$$
\begin{array}{ll}
(\mathcal{A},\mathcal{A}') \models \mathit{pres}(\mathcal{K}(t_1,...t_n)[C],\mathcal{K}(t'_1,...t'_n)[C'])[\eta,\eta'] \wedge & \\
(\mathcal{A}',\mathcal{A}'') \models \mathit{pres}(\mathcal{K}(t'_1,...t'_n)[C'],\mathcal{K}(t''_1,...t''_n)[C''])[\eta',\eta''] & \\
\quad \Leftrightarrow \mathcal{A} \models_d \mathcal{K}(t_1,...t_n)[C][\eta] \Leftrightarrow \mathcal{A}' \models_d \mathcal{K}(t'_1,...t'_n)[C'][\eta'] \wedge & \\
\quad\quad \mathcal{A}' \models_d \mathcal{K}(t'_1,...t'_n)[C'][\eta'] \Leftrightarrow \mathcal{A}'' \models_d \mathcal{K}(t''_1,...t''_n)[C''][\eta''] & \text{Defn. 5.2.1} \\
\quad \Rightarrow \mathcal{A} \models_d \mathcal{K}(t_1,...t_n)[C][\eta] \Leftrightarrow \mathcal{A}'' \models_d \mathcal{K}(t''_1,...t''_n)[C''][\eta''] & \text{Tab. 4.6, } C' \neq \_ \\
\quad \Leftrightarrow (\mathcal{A},\mathcal{A}'') \models \mathit{pres}(\mathcal{K}(t_1,...t_n)[C],\mathcal{K}(t''_1,...t''_n)[C''])[\eta,\eta''] & \text{Defn. 5.2.1}
\end{array}
$$

The proof for functional $\mathcal{K}$ goes similar — substitute $\Leftrightarrow$ by equality. □

Proof C.3.2 (Proof of Lemma 5.3.1) We prove the assumptions in consecutive order.

(1) We prove the assumption by induction on the length n of tr.

Case 1 ($n = 0$): Then $tr \equiv \langle(\mathcal{A}_0, t_0)\rangle$ and $\mathcal{A}_0 \models_d t_0 \in$ `existDObj` by definition such that $\mathit{state}(0,tr) \models_d \mathit{version}(0,tr) \in$ `existDObj`.

Case 2 ($n \mapsto n+1$): We conclude:

$$
\begin{array}{ll}
tr \in \mathit{traces}_{n+1}(\Delta) & \\
\Rightarrow \exists tr_n \in \mathit{traces}_n(\Delta), tr' \in \mathit{step}(\Delta) \bullet \{tr\} = \{tr_n\} \dot{\cup} \{tr'\} & \text{Defn. 5.3.1} \\
\Rightarrow \forall i \in \{1,...,n-1\} \bullet (& \\
\quad \mathit{state}(i,tr) = \mathit{state}(i,tr_n) \wedge \mathit{version}(i,tr) = \mathit{version}(i,tr_n) \wedge & \text{Defn. 5.3.1} \\
\quad \mathit{state}(i,tr_n) \models_d \mathit{version}(i,tr_n) \in \texttt{existDObj}) \wedge & \text{ind.hyp} \\
\quad \mathit{state}(n,tr) = \mathit{state}(0,tr') \wedge \mathit{version}(n,tr) = \mathit{version}(0,tr') \wedge & \text{Defn. 5.3.1} \\
\quad \mathit{state}(0,tr') \models_d \mathit{version}(0,tr') \in \texttt{existDObj} \wedge & \text{Defn. 3.3.7,Thm. 4.3.2} \\
\quad \mathit{state}(n+1,tr) = \mathit{state}(1,tr') \wedge \mathit{version}(n+1,tr) = \mathit{version}(1,tr') \wedge & \text{Defn. 5.3.1} \\
\quad \mathit{state}(1,tr') \models_d \mathit{version}(1,tr') \in \texttt{existDObj} & \text{Defn. 3.3.7,Thm. 4.3.2} \\
\Rightarrow \forall i \in \{1,...,n+1\} \bullet \mathit{state}(i,tr) \models_d \mathit{version}(i,tr) \in \texttt{existDObj} &
\end{array}
$$

(2) We show the stronger variant

$$\mathit{state}(j,tr) \models_d \texttt{oid}(\mathit{version}(i,tr)) \neq \texttt{oid}(\mathit{version}(j,tr)) \wedge \texttt{oid}(\mathit{version}(i,tr)) \in \texttt{usedOIDs}$$

for all $0 \leq i < j \leq n$ by induction on the length n of tr. Since $\mathit{src}(\Delta)$ is derived by basic transitions, the invariants of Cor. 3.3.1 hold, which will be used in the proof.

Case 1 ($n = 0$): Then $\mathit{state}(0,tr) \models_d \mathit{version}(0,tr) \in$ `existDObj` by definition. This implies $\mathit{state}(0,tr) \models_d \texttt{oid}(\mathit{version}(0,tr)) \in$ `usedOIDs` according to Cor. 3.3.1(1).

Case 2 ($n \mapsto n+1$): We conclude

$tr \in traces_{n+1}(\Delta)$
$\Rightarrow \exists tr_n \in traces_n(\Delta), tr' \in step(\Delta) \bullet \{tr\} = \{tr_n\} \fatsemi \{tr'\}$ Defn. 5.3.1
$\Rightarrow \forall i \in \{0,...,n-1\} \bullet (state(i,tr) = state(i,tr_n) \wedge version(i,tr) = version(i,tr_n)) \wedge$ Defn. 5.3.1
$\forall i < j \in \{0,...,n\} \bullet$
$(state(j,tr_n) \models_d \texttt{oid}(version(i,tr_n)) \neq \texttt{oid}(version(j,tr_n)) \wedge$ ind.hyp
$state(j,tr_n) \models_d \texttt{oid}(version(i,tr_n)) \in \texttt{usedOIDs})\ s \wedge$
$state(0,tr') \models_d version(n,tr_n) = version(0,tr') \wedge$
$state(n,tr) = state(0,tr') \wedge version(n,tr) = version(0,tr') \wedge$
$state(n+1,tr) = state(1,tr') \wedge version(n+1,tr) = version(1,tr') \wedge$
$\langle state(0,tr'), \texttt{trans}(version(0,tr') \mapsto version(1,tr')), state(1,tr') \rangle \subseteq \Delta$ Defn. 5.3.1

$\Rightarrow \forall i < j \in \{0,...,n\} \bullet$
$(state(j,tr) \models_d \texttt{oid}(version(i,tr)) \neq \texttt{oid}(version(j,tr)) \wedge$
$state(j,tr) \models_d \texttt{oid}(version(i,tr)) \in \texttt{usedOIDs})$
$state(n+1,tr) \models_d \texttt{oid}(version(n,tr)) \neq \texttt{oid}(version(n+1,tr)) \wedge$ (below)
$\forall id : \texttt{OID} \bullet (state(n,tr) \models_d id \in \texttt{usedOIDs} \Rightarrow state(n+1,tr) \models_d id \in \texttt{usedOIDs}) \wedge$
$state(n,tr) \models_d \texttt{oid}(version(n+1,tr)) \notin \texttt{usedOIDs}$
$\Rightarrow \forall i < j \in \{0,...,n+1\} \bullet$
$(state(j,tr) \models_d \texttt{oid}(version(i,tr)) \neq \texttt{oid}(version(j,tr)) \wedge$
$state(j,tr) \models_d \texttt{oid}(version(i,tr)) \in \texttt{usedOIDs})$

where the remaining implication follows by

$state(0,tr') \models_d version(n,tr_n) = version(0,tr') \wedge$
$state(n,tr) = state(0,tr') \wedge version(n,tr) = version(0,tr') \wedge$
$state(n+1,tr) = state(1,tr') \wedge version(n+1,tr) = version(1,tr') \wedge$
$\langle state(0,tr'), \texttt{trans}(version(0,tr') \mapsto version(1,tr')), state(1,tr') \rangle \subseteq \Delta$
$\Rightarrow state(n,tr) = state(0,tr') \wedge version(n,tr) = version(0,tr') \wedge$
$state(n+1,tr) = state(1,tr') \wedge version(n+1,tr) = version(1,tr') \wedge$
$state(0,tr') \models_d version(0,tr') \in \texttt{existDObj} \wedge$ Defn. 3.3.7
$\forall id : \texttt{OID} \bullet (state(0,tr') \models_d id \in \texttt{usedOIDs} \Rightarrow state(1,tr') \models_d id \in \texttt{usedOIDs}) \wedge$ Defn. 3.3.7
$state(1,tr') \models_d version(1,tr') \in \texttt{existDObj} \wedge$ Defn. 3.3.7
$state(0,tr') \models_d \texttt{oid}(version(1,tr')) \notin \texttt{usedOIDs}$ Lemma 3.3.4
$\Rightarrow state(n,tr) \models_d version(n,tr) \in \texttt{existDObj} \wedge$
$\forall id : \texttt{OID} \bullet (state(n,tr) \models_d id \in \texttt{usedOIDs} \Rightarrow state(n+1,tr) \models_d id \in \texttt{usedOIDs}) \wedge$
$state(n+1,tr) \models_d version(n+1,tr) \in \texttt{existDObj} \wedge$
$state(n,tr) \models_d \texttt{oid}(version(n+1,tr)) \notin \texttt{usedOIDs}$
$\Rightarrow state(n,tr) \models_d \texttt{oid}(version(n,tr)) \in \texttt{usedOIDs} \wedge$ Cor. 3.3.1(1)
$\forall id : \texttt{OID} \bullet (state(n,tr) \models_d id \in \texttt{usedOIDs} \Rightarrow state(n+1,tr) \models_d id \in \texttt{usedOIDs}) \wedge$
$state(n+1,tr) \models_d \texttt{oid}(version(n+1,tr)) \in \texttt{usedOIDs} \wedge$ Cor. 3.3.1(1)
$state(n,tr) \models_d \texttt{oid}(version(n+1,tr)) \notin \texttt{usedOIDs}$
$\Rightarrow state(n+1,tr) \models_d \texttt{oid}(version(n,tr)) \neq \texttt{oid}(version(n+1,tr)) \wedge$
$\forall id : \texttt{OID} \bullet (state(n,tr) \models_d id \in \texttt{usedOIDs} \Rightarrow state(n+1,tr) \models_d id \in \texttt{usedOIDs}) \wedge$
$state(n,tr) \models_d \texttt{oid}(version(n+1,tr)) \notin \texttt{usedOIDs}$

□

Proof C.3.3 (Proof of Lemma 5.4.1) We conclude
$dom(tr, x:\tau, \eta) = (\mathcal{A}, d) \wedge a \in d$
$\Rightarrow \eta \in Env(X, src(tr)) \wedge \mathcal{A} = src(tr) \wedge a \in d$ Defn. 5.4.1
$\Rightarrow \eta \in Env(X, \mathcal{A}) \wedge \mathcal{A} = src(tr) \wedge$
$a \in \{v \mid v \in \tau^{src(tr)}, ct \in CT^{min}_{src(tr)}(\Sigma, v), (src(tr), ct) \overset{t_d}{\rightsquigarrow} v\}$ Defn. 5.4.1
$\Rightarrow \eta \in Env(X, \mathcal{A}) \wedge a \in \tau^{\mathcal{A}}$

□

Proof C.3.4 (Proof of Lemma 5.4.2) We show the properties consecutively.

(1) Since all concept names are disjoint from all symbol names of Σ and X according to Tab. 4.3 and Defn. 4.3.3, $KT_\tau(X, \Sigma, KD) \cap T_\tau(X, \Sigma) = \emptyset$. Hence, the required property of Defn. 3.2.10 (extension operators) is satisfied.

(2) We conclude

$$
\begin{array}{ll}
dom(tr, x:\tau, \eta) = (\mathcal{A}, d) \wedge a \in d & \\
\Rightarrow \eta \in Env(X, src(\Delta)) \wedge \mathcal{A} = src(\Delta) \wedge a \in d & \text{Defn. 5.4.2} \\
\Rightarrow \eta \in Env(X, \mathcal{A}) \wedge \mathcal{A} = src(\Delta) \wedge & \\
\quad a \in \{v \mid v \in \tau^{src(\Delta)}, ct \in CT^{min}_{src(\Delta)}(\Sigma, v), (src(\Delta), ct) \overset{t_d}{\rightsquigarrow} v\} & \text{Defn. 5.4.2} \\
\Rightarrow \eta \in Env(X, \mathcal{A}) \wedge a \in \tau^{\mathcal{A}} &
\end{array}
$$

□

C.4 Proofs for Chap. 6

Proof C.4.1 (Proof of Lemma 6.2.1) We prove the assumption by induction on the structure of *op*.

Case 1 ($op \equiv \texttt{create}(f, \overline{t_i})$):
First, Σ includes `Top`. Hence, term values are unique (Lemma 3.2.3).
Second, $f(\texttt{oidToUse}, \overline{t_i}) \in T_\tau(X, \Sigma)$ since $f_{\texttt{OID} \times \overline{\tau_i} \to \tau} \in \mathcal{C}_\tau$ and $t_i \in T_{\tau_i}(X, \Sigma)$ for all $1 \leq i \leq n$ (Tab. 6.3). Hence, $\mathcal{V}^{\mathcal{A}}[\![f(\texttt{oidToUse}, \overline{t_i})]\!]\eta \in \tau^{\mathcal{A}} \cup \{\bot\}$ (Lemma 3.2.2).

Apart from that, rule application of *r_create* returns a unique update set U (cf. Tab. 6.2). If $U = \emptyset$, then $\mathcal{A}' = \mathcal{A}$ and $\Delta' = \Delta$. Otherwise, $\mathcal{A}' = \mathcal{A} + U$ and $\Delta' = \Delta; \langle \mathcal{A}, \mathsf{cre}(ct), \mathcal{A} + U \rangle, ct \in CT^{min}_{\mathcal{A}}(\Sigma, v)$. In both cases $\mathcal{A}'$ is uniquely determined (cf. Defn. 6.2.4).

Case 2 ($op \equiv \texttt{transform}(t_{src} \mapsto (f, \overline{t_i}))$): Similar.

Case 3 ($op \equiv \texttt{delete}(t)$): Similar.

□

Proof C.4.2 (Proof of Lemma 6.2.2) The `skip`-operation trivially leads to a consistent update set. We show consistency for the if-cases by deriving the update sets with the calculus of Tab. 6.2.

(1) The derivation

$$
\frac{\begin{array}{l}
[\![\texttt{oidToUse} := \texttt{nextID(oidToUse)}]\!]^{\mathcal{A}}_{\eta} = \\
\qquad \{((\texttt{oidToUse} \bullet \_) \mapsto \mathcal{V}^{\mathcal{A}}[\![\texttt{nextID(oidToUse)}]\!]\eta)\} \\
[\![\texttt{existDObj} := \{f(\texttt{oidToUse}, \overline{t_i})\}_s \cup \texttt{existDObj}]\!]^{\mathcal{A}}_{\eta} = \\
\qquad \{((\texttt{existDObj} \bullet \_) \mapsto \mathcal{V}^{\mathcal{A}}[\![\{f(\texttt{oidToUse}, \overline{t_i})\}_s \cup \texttt{existDObj}]\!]\eta)\} \\
[\![\texttt{usedOIDs} := \texttt{oidToUse} \cup \texttt{usedOIDs}]\!]^{\mathcal{A}}_{\eta} = \\
\qquad \{((\texttt{usedOIDs} \bullet \_) \mapsto \mathcal{V}^{\mathcal{A}}[\![\{\texttt{oidToUse}\}_s \cup \texttt{usedOIDs}]\!]\eta)\} \\
[\![\texttt{dep}(f(\texttt{oidToUse}, \overline{t_i})) := t_{sub}]\!]^{\mathcal{A}}_{\eta} = \{((\texttt{dep} \bullet \mathcal{V}^{\mathcal{A}}[\![f(\texttt{oidToUse}, \overline{t_i})]\!]\eta) \mapsto \mathcal{V}^{\mathcal{A}}[\![t_{sub}]\!]\eta)\}
\end{array}}{
\begin{array}{l}
[\![\texttt{par oidToUse} := \ldots \texttt{end par}]\!]^{\mathcal{A}}_{\eta} = \\
\qquad \left\{ \begin{array}{l}
((\texttt{oidToUse} \bullet \_) \mapsto \mathcal{V}^{\mathcal{A}}[\![\texttt{nextID(oidToUse)}]\!]\eta), \\
((\texttt{existDObj} \bullet \_) \mapsto \mathcal{V}^{\mathcal{A}}[\![\{f(\texttt{oidToUse}, \overline{t_i})\}_s \cup \texttt{existDObj}]\!]\eta), \\
((\texttt{usedOIDs} \bullet \_) \mapsto \mathcal{V}^{\mathcal{A}}[\![\{\texttt{oidToUse}\}_s \cup \texttt{usedOIDs}]\!]\eta), \\
((\texttt{dep} \bullet \mathcal{V}^{\mathcal{A}}[\![f(\texttt{oidToUse}, \overline{t_i})]\!]\eta) \mapsto \mathcal{V}^{\mathcal{A}}[\![t_{sub}]\!]\eta)
\end{array} \right\}
\end{array}
}
$$

shows the consistency for $\texttt{create}(f, \overline{t_i})$ (cf. Tab. 6.2.7).

(2) If the precondition is satisfied, a transformation $\texttt{transform}(t_{src} : \texttt{DObj} \mapsto (f, \overline{t_i}))$ yields the same update set as a creation operation $\texttt{create}(f, \overline{t_i})$, which proves the consistency.

(3) The derivation

$$\frac{\begin{array}{l}[\![\texttt{existDObj} := \texttt{existDObj}\backslash\{t\}_s]\!]^{\mathcal{A}}_{\eta} = \\ \quad \{((\texttt{existDObj} \bullet \_) \mapsto \mathcal{V}^{\mathcal{A}}[\![\texttt{existDObj}\backslash\{t\}_s]\!]\eta)\} \\ [\![\texttt{dep}(t) := t_{sub}]\!]^{\mathcal{A}}_{\eta} = \{((\texttt{dep} \bullet \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \mapsto \mathcal{V}^{\mathcal{A}}[\![\{\}]\!]\eta)\}\end{array}}{\begin{array}{l}[\![\texttt{par existDObj} := ... \texttt{end par}]\!]^{\mathcal{A}}_{\eta} = \\ \quad \left\{ \begin{array}{l} ((\texttt{existDObj} \bullet \_) \mapsto \mathcal{V}^{\mathcal{A}}[\![\texttt{existDObj}\backslash\{t\}_s]\!]\eta), \\ ((\texttt{dep} \bullet \mathcal{V}^{\mathcal{A}}[\![t]\!]\eta) \mapsto \mathcal{V}^{\mathcal{A}}[\![\{\}]\!]\eta) \end{array} \right\}\end{array}}$$

shows the consistency for $\texttt{delete}(t)$.

□

In the proof of Thm. 6.2.1 we use the Hoare-style proof logic developed in [Ton97]. We need the following three rules in order to cover the rules r_create, r_delete, and $r\_transform$.

(1) $\{\texttt{Cons}(U) \wedge \phi[\sigma_U]\}\ U\ \{\phi\}$ (**Upd**)

(2) $$\frac{\{\pi \wedge \phi\}\ P\ \{\psi\} \quad \{\pi \wedge \neg\phi\}\ Q\ \{\psi\}}{\{\pi\}\ \texttt{if}\ \phi\ \texttt{then}\ P\ \texttt{else}\ Q\ \texttt{end if}\ \{\psi\}}$$ (**If**)

(3) $$\frac{\phi \Rightarrow \pi\ \{\pi\}\ P\ \{\chi\}\ \chi \Rightarrow \psi}{\{\phi\}\ P\ \{\psi\}}$$ (**Conseq**)

Rule (1) deals with update sets U that contain simple updates of the form $s := t$ only. It corresponds to the usual assignment axiom of the Hoare calculus, but may assign new values not only to variables but to (dynamic) locations of the underlying algebra as well. In that, σ_U is the term substitution deriving from U. Having an update $s := t$, $\phi[\sigma_U]$ derives from ϕ by substituting all occurrences of those terms s by t simultaneously that equal t. Moreover, we deal with parallel updates, which may cause update sets to be inconsistent. Therefore, the explicit consistency requirement $\texttt{Cons}(U)$ is integrated into the rule's precondition.

Rules (2) and (3) handle if-clauses and logical consequence, respectively, similar to the usual Hoare-calculus.

Proof C.4.3 (Proof of Thm. 6.2.1) In order to prove the desired properties, we need the following invariants

$$inv_1 := \forall x \in \texttt{existDObj} \bullet \texttt{oid}(x) <_{id} \texttt{oidToUse}$$
$$inv_2 := \forall id \in \texttt{usedOIDs} \bullet id <_{id} \texttt{oidToUse}$$
$$inv_3 := \forall x, x' : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow (x' \in cont^{\mathcal{A}}_{\texttt{DObj}}(x) \Leftrightarrow x' \in \texttt{dep}(x) \vee x = x')$$

They will be proved for the single operations. Invariants inv_1 and inv_2 state that `oidToUse` steadily grows and does not store an ID that has already been used in the system. The latter states that $\texttt{dep}(t)$ contains the full object-valued content of existing digital objects t, except t itself.
Throughout the proof we show all properties (including the invariants) following the scheme

$$\frac{\phi\ \Rightarrow \{\psi[\sigma_U]\}\ U\ \{\psi\}}{\{\phi\}\ U\ \{\psi\}}$$

where ϕ and ψ either represent the invariant or represent the respective pre-and post-conditions of the basic state change. U is the update set that derives from the respective basic operation when the pre-condition is satisfied.

Object creation: From Lemma 6.2.2 we know that the update set for the creation operation $\texttt{create}(f, \overline{t_i})$ is consistent. Hence, we can omit it in the preconditions. Moreover, $\texttt{create}(f, \overline{t_i})$ results in the ASM rule

$$r\_create(f, \overline{t_i}, T^{\leq}_{\texttt{DObj}}(\emptyset, ct)\backslash\{ct\}).$$

There, $\mathcal{A} \models ct = f(\texttt{oidToUse}, \overline{t_i})$, which will be referred to by (*) in the proofs when needed. Since the result sequence Δ is extended, the if-case of the rule declaration *r_create* is used according to Defn. 6.2.8. The update set of *r_create*, thus corresponds to:

$$U := \left\{ \begin{array}{l} \texttt{oidToUse} := \texttt{nextID}(\texttt{oidToUse}) \\ \texttt{existDObj} := \{f(\texttt{oidToUse}, \overline{t_i}\}_s \cup \texttt{existDObj} \\ \texttt{usedOIDs} := \{\texttt{oidToUse}\}_s \cup \texttt{usedOIDs} \\ \texttt{dep}(f(\texttt{oidToUse}, \overline{t_i})) := t_{sub} \end{array} \right\},$$

We first prove the invariants.

inv_1 : We conclude

$\forall x \in \texttt{existDObj} \bullet \texttt{oid}(x) <_{id} \texttt{oidToUse}$
$\Rightarrow \forall x \in \texttt{existDObj} \bullet \texttt{oid}(x) <_{id} \texttt{nextID}(\texttt{oidToUse}) \wedge$
$\quad \texttt{oidToUse} <_{id} \texttt{nextID}(\texttt{oidToUse})$ Defn. 3.3.2
$\Rightarrow \forall x \in \texttt{existDObj} \cup \{f(\texttt{oidToUse}, \overline{t_i})\} \bullet \texttt{oid}(x) <_{id} \texttt{nextID}(\texttt{oidToUse})$
$\equiv \forall x \in \texttt{existDObj} \bullet \texttt{oid}(x) <_{id} \texttt{oidToUse}[\sigma_U]$

inv_2 : We conclude

$\forall id \in \texttt{usedOIDs} \bullet \texttt{oid}(x) <_{id} \texttt{oidToUse}$
$\Rightarrow \forall id \in \texttt{usedOIDs} \bullet id <_{id} \texttt{nextID}(\texttt{oidToUse}) \wedge$
$\quad \texttt{oidToUse} <_{id} \texttt{nextID}(\texttt{oidToUse})$ Defn. 3.3.2
$\Rightarrow \forall x \in \texttt{usedOIDs} \cup \{\texttt{oidToUse}\} \bullet \texttt{oid}(x) <_{id} \texttt{nextID}(\texttt{oidToUse})$
$\equiv \forall x \in \texttt{usedOIDs} \bullet id <_{id} \texttt{oidToUse}[\sigma_U]$

inv_3 : Since $inv_3[\phi_U]$ is different for $x = f(\texttt{oidToUse}, \overline{t_i})$ and $x \neq f(\texttt{oidToUse}, \overline{t_i})$, we distinguish these two cases.

Case $(x = f(\texttt{oidToUse}, \overline{t_i}))$:

Then

$$inv_3[\sigma_U] \equiv f(\texttt{oidToUse}, \overline{t_i}) \in \texttt{existDObj} \cup \{f(\texttt{oidToUse}, \overline{t_i})\} \Rightarrow (x' \in cont^{\mathcal{A}}_{\texttt{DObj}}(f(\texttt{oidToUse}, \overline{t_i})) \Leftrightarrow x' \in t_{sub} \vee x = x')$$

and we conclude

$f(\texttt{oidToUse}, \overline{t_i}) \in \texttt{existDObj} \cup \{f(\texttt{oidToUse}, \overline{t_i})\}$
$\Rightarrow \texttt{True}$
$\Rightarrow t_{sub} = T^{\leq}_{\texttt{DObj}}(\emptyset, ct)\backslash\{ct\}$ Defn. 6.2.8
$\Rightarrow (x' \in cont^{\mathcal{A}}_{\texttt{DObj}}(ct) \Leftrightarrow x' \in t_{sub} \vee ct = x')$ Defn. 3.3.3
$\Rightarrow x' \in cont^{\mathcal{A}}_{\texttt{DObj}}(f(\texttt{oidToUse}, \overline{t_i})) \Leftrightarrow x' \in t_{sub} \vee x = x'$ (*)

Case $(x \neq f(\texttt{oidToUse}, \overline{t_i}))$:

Then

$$inv_3[\sigma_U] \equiv x \in \texttt{existDObj} \cup \{f(\texttt{oidToUse}, \overline{t_i})\} \Rightarrow (x' \in cont^{\mathcal{A}}_{\texttt{DObj}}(x) \Leftrightarrow x' \in \texttt{dep}(x) \vee x = x')$$

and we conclude

$x \in \texttt{existDObj} \cup \{f(\texttt{oidToUse}, \overline{t_i})\}$
$\Rightarrow x \in \texttt{existDObj}$ $x \neq f(\texttt{oidToUse}, \overline{t_i})$
$\Rightarrow x' \in cont^{\mathcal{A}}_{\texttt{DObj}}(x) \Leftrightarrow x' \in \texttt{dep}(x) \vee x = x'$ prereq.

Now, we are ready to show that $\texttt{create}(ct)$ respects the pre- and post conditions of Defn. 3.3.7. In particular, we show the following properties:

(1) $\mathcal{A} \models \forall x : \texttt{DObj} \bullet x \in t_{sub} \Rightarrow x \in \texttt{existDObj}$ implies $\mathcal{A} \models t' \in \texttt{existDObj}$ for all $t' \in T^{\leq}_{\texttt{DObj}}(\emptyset, ct) \backslash \{ct\}$.

(2) $ct \in \texttt{existDObj} \wedge \texttt{oid}(ct) \in \texttt{usedOIDs}[\sigma_U]$.

(3) $t \in \texttt{existDObj} \Rightarrow (t \in \texttt{existDObj} \wedge t \neq ct[\sigma_U])$ for arbitrary $t \in GT_{\texttt{DObj}}(sp(\Sigma))$.

(3') $t \notin \texttt{existDObj} \Rightarrow (t \notin \texttt{existDObj} \vee t = ct[\sigma_U])$ for arbitrary $t \in GT_{\texttt{DObj}}(sp(\Sigma))$.

(4) $id \in \texttt{usedOIDs} \Rightarrow (id \in \texttt{usedOIDs} \wedge id \neq \texttt{oid}(ct)[\sigma_U])$ for arbitrary $id \in GT_{\texttt{OID}}(sp(\Sigma))$.

(4') $id \notin \texttt{usedOIDs} \Rightarrow (id \notin \texttt{usedOIDs} \vee id = \texttt{oid}(ct)[\sigma_U])$ for arbitrary $id \in GT_{\texttt{OID}}(sp(\Sigma))$.

Properties (1) and (2) assure requirements (1) and (2), respectively, of Defn. 3.3.7. There, the first part of property (1) is the if-condition of the rule *r_create*. Properties (3),(3') and (4),(4') assure requirements (3) and (4), respectively. Requirement (5) of Defn. 3.3.7 always holds as ASM rule applications may change dynamic functions only.

(1) According to Defn. 6.2.8 $t_{sub} = T^{\leq}_{\texttt{DObj}}(\emptyset, ct) \backslash \{ct\}$, which proves the assumption.

(2) Since

$$ct \in \texttt{existDObj} \wedge \texttt{oid}(ct) \in \texttt{usedOIDs}[\sigma_U] \equiv ct \in \texttt{existDObj} \cup \{f(\texttt{oidToUse}, \overline{t_i})\} \wedge \texttt{oid}(ct) \in \{\texttt{oidToUse}\}_s \cup \texttt{usedOIDs}$$

and $\texttt{oid}(ct) = \texttt{oidToUse}$ due to (*), the assumption $ct \in \texttt{existDObj} \wedge \texttt{oid}(ct) \in \texttt{usedOIDs}[\sigma_U]$ directly follows.

(3) Since

$$t \in \texttt{existDObj} \wedge t \neq ct[\sigma_U] \equiv t \in \texttt{existDObj} \cup \{f(\texttt{oidToUse}, \overline{t_i})\} \wedge t \neq ct,$$

$\texttt{oidToUse} = \texttt{oid}(ct)$ due to (*), $t \in \texttt{existDObj} \Rightarrow \texttt{oid}(t) <_{id} \texttt{oidToUse}$ (inv_1), and, hence, $\texttt{oid}(t) \neq \texttt{oidToUse}$, the implication $t \in \texttt{existDObj} \Rightarrow (t \in \texttt{existDObj} \wedge t \neq ct[\sigma_U])$ directly follows.

(3') Since

$$t \notin \texttt{existDObj} \vee t = ct[\sigma_U] \equiv t \notin \texttt{existDObj} \cup \{f(\texttt{oidToUse}, \overline{t_i})\} \vee t = ct$$

and $ct = f(\texttt{oidToUse}, \overline{t_i})$ due to (*), the implication $t \notin \texttt{existDObj} \Rightarrow (t \notin \texttt{existDObj} \vee t = ct[\sigma_U])$ directly follows.

(4) Since

$$id \in \texttt{usedOIDs} \wedge id \neq \texttt{oid}(ct)[\sigma_U] \equiv id \in \texttt{usedOIDs} \cup \{\texttt{oidToUse})\} \wedge id \neq \texttt{oid}(ct),$$

$\texttt{oidToUse} = \texttt{oid}(ct)$ due to (*), $id \in \texttt{usedOIDs} \Rightarrow id <_{id} \texttt{oidToUse}$ (inv_2), and, hence, $id \neq \texttt{oidToUse}$, the implication $id \in \texttt{usedOIDs} \Rightarrow (id \in \texttt{usedOIDs} \wedge id \neq \texttt{oid}(ct)[\sigma_U])$ directly follows.

(4') Since

$$id \notin \texttt{usedOIDs} \vee t = ct[\sigma_U] \equiv id \notin \texttt{usedOIDs} \cup \{\texttt{oidToUse}\} \vee id = \texttt{oid}(ct)$$

and (*), the implication $id \notin \texttt{usedOIDs} \Rightarrow (id \notin \texttt{usedOIDs} \vee t = ct[\sigma_U])$ directly follows.

Object transformation: From Lemma 6.2.2 we know that the update set for the transformation operation $\texttt{transform}(t_{src} : \texttt{DObj} \mapsto (f, \overline{t_i}))$ is consistent. Again, we only consider the if-case. Then this operation yields the same update set U as for $\texttt{create}(f, \overline{t_i})$. Since the if-condition of an object transformation implies the one for a corresponding creation operation, the proves carry over using rule **Conseq**.

Object deletion: From Lemma 6.2.2 we know that the update set for the delete operation $\texttt{delete}(ct)$ is consistent. Hence, we can omit it in the preconditions. Again, we use $ct = f(\texttt{oidToUse}, \overline{t_i})$ and refer to it by (*) in the proofs. Let

$$U := \left\{ \begin{array}{l} \texttt{existDObj} := \texttt{existDObj} \backslash \{ct\}_s \\ \texttt{dep}(ct) := \{\} \end{array} \right\}$$

denote the corresponding rule set.

We start by showing validity of inv_1, inv_2, inv_3. Since always $\texttt{existDObj}\backslash\{ct\}_s \subseteq \texttt{existDObj}$ and $\texttt{usedOIDs}$ is not affected at all by these rules, it is easy to see that inv_1 and inv_2 hold. The proof for inv_3 is given as follows:

inv_3 : Since $inv_3[\phi_U]$ is different for $x = ct$ and $x \neq ct$, we distinguish these two cases.

Case $(x = ct)$:
Then

$$\begin{aligned} inv_3[\sigma_U] \equiv\ & ct \in \texttt{existDObj}\backslash\{ct\}_s \Rightarrow \\ & (x' \in cont^{\mathcal{A}}_{\texttt{DObj}}(ct) \Leftrightarrow x' \in \{\} \vee ct = x') \end{aligned}$$

and we conclude
$ct \in \texttt{existDObj}\backslash\{ct\}_s \Rightarrow \texttt{False}$
$\Rightarrow (x' \in cont^{\mathcal{A}}_{\texttt{DObj}}(ct) \Leftrightarrow x' \in \{\} \vee ct = x')$

Case $(x \neq ct)$:
Then

$$\begin{aligned} inv_3[\sigma_U] \equiv\ & x \in \texttt{existDObj}\backslash\{ct\}_s \Rightarrow \\ & (x' \in cont^{\mathcal{A}}_{\texttt{DObj}}(x) \Leftrightarrow x' \in \texttt{dep}(x) \vee x = x') \end{aligned}$$

and we conclude
$x \in \texttt{existDObj}\backslash\{ct\}_s \Rightarrow x \in \texttt{existDObj}$
$\Rightarrow x' \in cont^{\mathcal{A}}_{\texttt{DObj}}(x) \Leftrightarrow x' \in \texttt{dep}(x) \vee x = x'$ prereq.

Now, we show that $\texttt{delete}(ct)$ respects the pre- and post conditions of Defn. 3.3.7. We do this in analogy to object creation $\texttt{cre}(ct)$. In particular, we show the following properties:

(1) $\mathcal{A} \models ct \in \texttt{existDObj} \wedge \forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow ct \notin \texttt{dep}(x)$ implies $\mathcal{A} \models t \in \texttt{existDObj} \Rightarrow ct \notin cont^{\mathcal{A}}_{\texttt{DObj}}(t) \vee t = ct$ for all $t \in GT_{\texttt{DObj}}(sp(\Sigma))$.

(2) $\mathcal{A} \models ct \in \texttt{existDObj} \wedge \forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow ct \notin \texttt{dep}(x)$ implies $\mathcal{A} \models ct \in \texttt{existDObj}$.

(3) $t \in \texttt{existDObj} \wedge t \neq ct \Rightarrow (t \in \texttt{existDObj}[\sigma_U])$ for arbitrary $t \in GT_{\texttt{DObj}}(sp(\Sigma))$.

(3') $t \notin \texttt{existDObj} \vee t = ct \Rightarrow (t \notin \texttt{existDObj}[\sigma_U])$ for arbitrary $t \in GT_{\texttt{DObj}}(sp(\Sigma))$.

(4) $id \in \texttt{usedOIDs} \Leftrightarrow (id \in \texttt{usedOIDs}[\sigma_U])$ for arbitrary $id \in GT_{\texttt{OID}}(sp(\Sigma))$.

Properties (1), (2), and (4) assure requirements (1), (2), and (4) of Defn. 3.3.7, respectively. There, the first part of properties (1),(2) is the if-condition of the rule *r_delete*. Properties (3),(3') assure requirement (3). Requirement (5) of Defn. 3.3.7 always holds as ASM rule applications may change dynamic functions only.

(1) Assume $ct \in \texttt{existDObj} \wedge \forall x : \texttt{DObj} \bullet x \in \texttt{existDObj} \Rightarrow ct \notin \texttt{dep}(x)$. Then we conclude:

$$\begin{aligned} \mathcal{A} \models t \in \texttt{existDObj} &\Rightarrow \mathcal{A} \models ct \notin \texttt{dep}(t) && \text{prereq.} \\ &\Rightarrow \mathcal{A} \models ct \notin cont^{\mathcal{A}}_{\texttt{DObj}}(t) \vee t = ct && inv_3 \end{aligned}$$

(2) This trivially holds.

(3) We conclude

$$\begin{aligned} t \in \texttt{existDObj} \wedge t \neq ct &\Rightarrow t \in \texttt{existDObj}\backslash\{ct\}_s \\ &\equiv t \in \texttt{existDObj}[\sigma_U] \end{aligned}$$

(3') We conclude

$$\begin{aligned} t \notin \texttt{existDObj} \vee t = ct &\Rightarrow t \notin \texttt{existDObj}\backslash\{ct\}_s \\ &\equiv t \notin \texttt{existDObj}[\sigma_U] \end{aligned}$$

(4) We conclude

$$\begin{aligned} id \in \texttt{usedOIDs} &\Leftrightarrow id \in \texttt{usedOIDs} \\ &\equiv id \in \texttt{usedOIDs}[\sigma_U] \end{aligned}$$

□

Proof C.4.4 (Proof of Lemma 6.3.1) We merely show that the new Boolean-valued functions respect overloading. It is easy to see that validity of all other consistency conditions is preserved when translating Σ-algebras to $gen(\Sigma)$-algebras. Suppose

$$f_{\overline{\tau_{1,i}}\to\texttt{Bool}}, f_{\overline{\tau_{2,j}}\to\texttt{Bool}} \in \mathcal{F}_{gen(\Sigma)} \text{ and } f_{\overline{\tau_{1,i}}}, f_{\overline{\tau_{2,j}}} \in \mathcal{P}_\Sigma.$$

We denote $gen(\Sigma, \mathcal{A})$ by $\mathcal{A}'$ and conclude:

$$\begin{aligned} & f_{\overline{\tau_{1,i}}\to\texttt{Bool}} \cong f_{\overline{\tau_{2,j}}\to\texttt{Bool}} \wedge \overline{\tau_k}{<}\overline{\tau_{1,i}}, \overline{\tau_{2,j}} \wedge \overline{v_k} \in \overline{\tau_k^{\mathcal{A}'}} \\ & \Rightarrow f_{\overline{\tau_{1,i}}} \cong f_{\overline{\tau_{2,j}}} \wedge \overline{\tau_k}{<}\overline{\tau_{1,i}}, \overline{\tau_{2,j}} \wedge \overline{v_k} \in \overline{\tau_k^{\mathcal{A}}} && \text{prereq.} \\ & \Rightarrow \overline{v_k} \in f^{\mathcal{A}}_{\overline{\tau_{1,i}}} \Leftrightarrow \overline{v_k} \in f^{\mathcal{A}}_{\overline{\tau_{2,j}}} && \text{Defn. 3.2.3} \\ & \Rightarrow f^{\mathcal{A}'}_{\overline{\tau_{1,i}}\to\texttt{Bool}}(\overline{v_k}) = \texttt{True}^{\mathcal{A}'} \Leftrightarrow f^{\mathcal{A}'}_{\overline{\tau_{2,j}}\to\texttt{Bool}}(\overline{v_k}) = \texttt{True}^{\mathcal{A}'} \wedge && \text{Defn. 6.3.1} \\ & \quad f^{\mathcal{A}'}_{\overline{\tau_{1,i}}\to\texttt{Bool}}(\overline{v_k}) = \texttt{False}^{\mathcal{A}'} \Leftrightarrow f^{\mathcal{A}'}_{\overline{\tau_{2,j}}\to\texttt{Bool}}(\overline{v_k}) = \texttt{False}^{\mathcal{A}'} && \text{Defn. 6.3.1} \end{aligned}$$

□

Proof C.4.5 (Proof of Thm. 6.3.1) We prove the assumption by induction on the structure of e.

Case 1 ($e \in T_\tau(X, gen(\Sigma) \cup (\emptyset, \emptyset, \emptyset, \emptyset, \mathcal{F}_{def}))$):
We distinguish two cases.

Case 1.1 ($e \in T_\tau(X, gen(\Sigma))$):
This case is covered by rule **Σ-terms 1**. There, $\mathcal{A}' = \mathcal{A}$ and $\Delta' = \Delta$. Also, $\overset{t}{\rightsquigarrow}$ derives unique term values for $gen(\Sigma)$-terms (Lemma 6.3.1, Lemma 3.2.2).

Case 1.2 ($e \notin T_\tau(X, gen(\Sigma))$):
This case is covered by rules **Σ-terms 2** and **Σ-terms 3**. Since e is not in $T_\tau(X, gen(\Sigma))$ it contains at least one function symbol of $N_{\mathcal{F}_{def}}$. Now, either $e \equiv f(t_1, ...t_n)$, $f \in N_{\mathcal{F}_{def}}$ or $e \equiv f(t_1, ..., t_n)$, $f \notin N_{\mathcal{F}_{def}}$ $(n \geq 0)$.

Case 1.2.1 ($e \equiv f(t_1, ...t_n),\ f \in N_{\mathcal{F}_{def}}$):
Suppose $f_{\overline{\tau_i} \to \tau} \in \mathcal{F}_{def}$. We apply rule **Σ-terms 2**. Since $e \in T_\tau(X, gen(\Sigma) \cup (\emptyset, \emptyset, \emptyset, \emptyset, \mathcal{F}_{def}))$, FD is suitable for $\mathcal{F}_{def}$, and $\mathcal{F}_{def}$ is not overloaded, there is a unique function definition $f(x_1, ..., x_n) = e' \in FD$ such that $e' \in E_\tau(X, \Sigma, \mathcal{F}_{def})$ (Defn. 6.3.2, Defn. 6.3.3). Furthermore, the terms $t_1, ..., t_n$ and the expression e' are shorter than $f(t_1, ..., t_n)$. By induction hypothesis they, thus, return unique $\mathcal{A}_{i+1}, v_i \in \tau^{\mathcal{A}_{i+1}} \cup \{\bot\}$ $(i \in \{1, ..., n\})$ and $\mathcal{A}', v \in \tau^{\mathcal{A}'} \cup \{\bot\}$ using $\overset{mig}{\rightsquigarrow}$. As the overall result for e is set to $(\mathcal{A}', \Delta', v)$, the assumption holds.

Case 1.2.2 ($e \equiv f(t_1, ...t_n),\ f \notin N_{\mathcal{F}_{def}}$):
Rule **Σ-terms 3** is applied. The result follows similar to case 1.2.1.

Case 2 ($e \in \mathcal{BO}p_\tau(X, \Sigma)$):
The rule **Basic operations** is applied. Since $\overset{op}{\rightsquigarrow}$ satisfies the property (Lemma 6.2.1), $\overset{mig}{\rightsquigarrow}$ does as well.

Case 3 ($e \equiv \texttt{let}\ x_1 = e_1 ... x_n = e_n\ \texttt{in}\ e_{n+1}$):
The rule **Let expressions** is applied. The expressions $e_1, ..., e_{n+1}$ are shorter than e. By induction hypothesis they, thus, return unique $\mathcal{A}_2, v_1 \in \tau^{\mathcal{A}_2} \cup \{\bot\}$ and $\mathcal{A}_{i+1}, v_i \in \tau^{\mathcal{A}_{i+1}} \cup \{\bot\}$ $(i \in \{2, ..., n+1\})$ using $\overset{mig}{\rightsquigarrow}$. As the overall result for e is set to $(\mathcal{A}_{n+2}, \Delta_{n+2}, v_{n+1})$, the assumption holds.

Case 4 ($e \equiv \texttt{if}\ e_B\ \texttt{then}\ e_0\ \texttt{else}\ e_1$):
Depending on the value of e_B, which is uniquely determined by induction hypothesis, exactly one of the three variants of rule **if-then-else expressions** is applicable. Hence, the result is set to $\bot$, the result for e_0, or the result for e_1, respectively, which proves the assumption (ind.hyp.).

Case 5 ($e \equiv \texttt{cast}(e', \tau)$):
Follows similarly.

□

C.5 Proofs for Chap. 7

Proof C.5.1 (Proof of Lemma 7.2.1) It is easy to see that $sep(A, G)$ and $red(A, G)$ are composable. We show that the components of $sep(A, G); red(A, G)$ equal their counterpart in G. There, we abbreviate $sep(A, G); red(A, G)$ by $G_;$.

($N_{G_;} = N_G$):
$N_{G_;} = reach_G(A) \cup N_G \backslash \{A\} = \{A\} \cup N_G \backslash \{A\} = N_G$.

($\Omega_{G_;} = \Omega_G$)
$\Omega_{G_;} = ((\Omega_G \cup \{A\}) \cup \Omega_G) \backslash \{A\} = \Omega_G$ (as $A \notin \Omega_G$).

($P_{G_;} = P_G$):
$P_{G_;} = (P_G \backslash \{A \to \alpha \mid A \to \alpha \in P_G\}) \cup \{X \to \alpha \mid X \to \alpha \in P_G, X \in reach_G(A)\} =$
$(P_G \backslash \{A \to \alpha \mid A \to \alpha \in P_G\}) \cup \{A \to \alpha \mid A \to \alpha \in P_G\} =$
P_G (since $A \in reach_G(A)$).

($S_{G_;} = S_G$): This directly follows from the definition.

□

Proof C.5.2 (Proof of Lemma 7.4.1) Given a deterministic HA and a word w. We prove the assumption by induction on the length l of a *maximal* segmentation $w_0....w_{l-1}$ of w such that there is a run of HA over w; these segmentations exist since we forbid ϵ-transitions.
Case 1 ($l = 1$): Then $w_0 = w \in \mathit{symbols}^e(HA)$. Given two runs q_0wq_1 and q_0wq_1'. Since HA is branching deterministic, $q_1 = q_1'$ (Defn. 7.4.2(1)).
Case 2 ($l > 1$):. Given runs $q_0w_1q_1....q_n$ and $q_0w_1'q_1'...q_m$ of HA over w, and let $w = w_1w_2$ and $w = w_1'w_2'$. Since $q_0w_1q_1$, there is a transition $(q_0, l_1, q_1) \in \delta_{HA}$ that can fire on w_1. Analogously, there is a transition $(q_0, l_2, q_1') \in \delta_{HA}$ that can fire on w_1'. Since HA is sequentially deterministic, $w_1' = w_1$ (Defn. 7.4.2(1)). As HA is branching deterministic, $q_1 = q_1'$ and $l_1 = l_1'$. Since w_2 and w_2' (thus, lengths of maximal segmentations for w_1 and w_2) are shorter and both runs start in the same state (q_1), we can apply the induction hypothesis. We conclude that the overall run is unique. □

Proof C.5.3 (Proof of Lemma 7.4.2) Derivations in G always have the following form:

$$S \overset{G}{\Rightarrow} \mathtt{a_0}A_1 \overset{G}{\Rightarrow} ... \overset{G}{\Rightarrow} \mathtt{a_0}...\mathtt{a_{n-1}}A_n \overset{G}{\Rightarrow} \mathtt{a_0}...\mathtt{a_n}$$

From the construction in Defn. 7.4.3 we see that $S\mathtt{a_0}A_1...\mathtt{a_{n-1}}A_n\mathtt{a_n}X$ is an accepting run in $HA(G)$. Analogously, runs over $HA(G)$ are translated to derivations in G. □

Proof C.5.4 (Proof of Thm. 7.4.1) We show the two directions separately and abbreviate $G;G'$ by $G_;$ and $HA_G[S' \uparrow HA'_G]$ by $HA\uparrow$.

$L(G_;) \subseteq L(HA\uparrow)$
Given a word w and a derivation

$$S \overset{G_;}{\Rightarrow} \alpha_1\beta_1 \overset{G_;}{\Rightarrow} ... \overset{G_;}{\Rightarrow} \alpha_{n-1}\beta_{n-1} \overset{G_;}{\Rightarrow} w.$$

Without loss of generality, we assume that this derivation is left-recursive (i.e., innermost non-terminals are resolved first, cf. [HMU06]). Both, G and G' are right-linear. Hence we can assume that α_i is a (possibly empty) sequence of terminals; we denote α_i by w_i. Also, β_i contains none, one, or two non-terminals and no terminals. In particular, occurrences of $S'\beta$ resolve S' in G' first and somewhen lead to $w'\beta$, where $w' \in \Omega^*_{G_;}$ (β_i stays unchanged until S' is completely resolved).
We reduce the above derivation appropriately such that it becomes an accepting run from S to X in $HA\uparrow$. For this purpose, we apply the following procedure, where $A_0 = S$ and $\mathit{result} = \langle\rangle$.

(1) Set $i = 1$. Go to (2).

(2) If $\beta_i = \epsilon$ goto (3). Otherwise take the derivation step $w_{i-1}\beta_{i-1} \overset{G_;}{\Rightarrow} w_i\beta_i$ and distinguish the following patterns:

Case 1 ($w_{i-1}A_{i-1} \overset{G_;}{\Rightarrow} w_iS'\beta_i' \quad (\beta_i \equiv S'\beta_i')$): Then there is a derivation $S' \overset{G'}{\Rightarrow} w'_{i+i}... \overset{G'}{\Rightarrow} w'_k$ in G' such that $w_iS'\beta_i' \overset{G_;}{\Rightarrow} w_{i+1}\beta_{i+1}... \overset{G_;}{\Rightarrow} w_k\beta_k$ is a derivation in $G_;$, $\beta_j = \beta_i$ for $i \leq j \leq k$ and $w_iw'_j = w_j$ for $i < j \leq k$ (in particular, $w_iw'_k = w_k$). This derivation exists since w is accepted, the overall derivation is left-recursive, and G, G' are composable; all productions of shared non-terminals of G and G'

are *equal* (cf. Defn. 7.2.3). We conclude that $HA(G')$ accepts w'_k (Lemma 7.4.2). Also, β_k is either empty or a non-terminal A_k. Hence, either (A_{i-1}, S', X) or (A_{i-1}, S', A_k) is a transition in HA_G (Defn. 7.4.3). Thus, either $(A_{i-1}, HA(G'), X)$ or $(A_{i-1}, HA(G'), A_k)$ is a hierarchical transition in $HA\uparrow$. Since $HA(G')$ accepts w'_k, this transition can fire and is a permissible step in $HA\uparrow$ (Defn. 7.4.1).

Append $A_{i-1}w'_kX$ and $A_{i-1}w'_kA_k$, respectively, to *result*; set i to $k+1$ and go to (2).

Case 2 ($w_{i-1}A_{i-1} \overset{G_;}{\Rightarrow} w_i\beta_i,\ \beta_i \in \{\epsilon\} \cup (N_;\backslash\{S'\})$): Then there is w' such that $w_i = w_{i-1}w'$ and $A_{i-1} \to w'$ is a production in $G_;$. Since $A_{i-1} \in N_;$, this is also a production in G (Defn. 7.2.3). If $\beta_i = \epsilon$, (A_{i-1}, w', X) is a transition in $HA(G)$. If $\beta_i = A_i, A_i \in N_;\backslash\{S'\}$, (A_{i-1}, w', A_i) is a transition in $HA(G)$ (Defn. 7.4.3). Since $w' \neq S'$, either of the transitions is also in $HA\uparrow$ and can fire. Thus, $A_{i-1}w'X$ and $A_{i-1}w'A_i$, respectively, is a permissible step in $HA\uparrow$.

Append $A_{i-1}w'X$ and $A_{i-1}w'A_i$, respectively, to *result*; set i to $i+1$ and go to (2).

(3) Stop.

$L(HA\uparrow) \subseteq L(G;)$

In the prove just shown we have *reduced* derivations in $G_;$ such that they become accepting runs in $HA\uparrow$. Now, we have to *expand* accepting runs of $HA\uparrow$ such that they are valid derivations in $G_;$. Given an accepting run

$$Sw_1A_1w_2....A_{n-1}w_nX$$

of $HA\uparrow$ over $w = w_1....w_n$. Then there are transitions

$$\delta_1 = (A_0, l_1, A_1), \delta_2 = (A_1, l_2, A_2), ..., \delta_n = (A_{n-1}, l_n, A_n)\ \ (A_0 = S, A_n = X)$$

such that δ_i that can fire on w_i in $HA\uparrow$.

For $(1 \leq i \leq n)$ we set the *initial derivation* $d(i)$ as follows:

$$d(i) := \begin{cases} A_{i-1} \overset{G_;}{\Rightarrow} \alpha_i^{k(i)}A'_i, & l_i = w_i \\ A_{i-1} \overset{G_;}{\Rightarrow} S'A_i \overset{G_;}{\Rightarrow} \alpha_i^1B^1A'_i \overset{G_;}{\Rightarrow} ... \overset{G_;}{\Rightarrow} \alpha_i^{k(i)-1}B^{k(i)-1}A'_i \overset{G_;}{\Rightarrow} \alpha_i^{k(i)}A'_i, & l_i = HA(G') \end{cases}$$

There, $\alpha_i^{k(i)} = w_i, A'_i = A_i$ for $1 \leq i < n$, and $A'_n = \epsilon$; $k(i) \in \mathbb{N}$ denotes the derivation length for w_i in $G_;$. If $l_i = w_i$ then $k(i) = 1$. Otherwise, $k(i)$ is the derivation length for w_i in G'.

$d(i)$ derives w_i in G, if step i of the accepting run corresponds to firing a basic transition in $HA\uparrow$; this automatically is a derivation in $G_;$. However, if step i is the result of firing a *hierarchical* transition (then $l_i = HA(G')$), $d(i)$ bases on the derivation

$$S' \overset{G'}{\Rightarrow} \alpha_i^1B^1 \overset{G'}{\Rightarrow} ... \overset{G'}{\Rightarrow} \alpha_i^{k(i)-1}B^{k(i)-1} \overset{G'}{\Rightarrow} \alpha_i^{k(i)}$$

for w_i in G'. It exists since $HA(G')$ accepts w_i. As $HA\uparrow$ has been constructed from G by substituting S' with $HA_{G'}$ (Defn. 7.4.3), $HA(G)$ has a transition (A_{i-1}, S', A_i).

Hence, G has a production $A_{i-1} \to S'A'_i$. Therefore, the derivation sequence $A_{i-1} \overset{G_;}{\Rightarrow} S'A_i \overset{G_;}{\Rightarrow} \alpha_i^1 B^1 A'_i \overset{G_;}{\Rightarrow} ... \overset{G_;}{\Rightarrow} \alpha_i^{k(i)-1} B^{k(i)-1} A'_i \overset{G_;}{\Rightarrow} \alpha_i^{k(i)} A'_i$ is indeed valid for $G_;$.
Up to now, the $d(i)$ are valid derivations for w_i in $G_;$. We remain to extend the overall derivation such that $w = w_1...w_n$ is derived. We define $w_0^< = \epsilon$, $w_i^< = w_0...w_{i-1}$ and set

$$d'(i) := w_i^< A_{i-1} \overset{G_;}{\Rightarrow} w_i^< \alpha_i^1 B^1 A'_i \overset{G_;}{\Rightarrow} ... \overset{G_;}{\Rightarrow} w_i^< \alpha_i^{k(i)-1} B^{k(i)-1} A'_i \overset{G_;}{\Rightarrow} w_i^< \alpha_i^{k(i)} A'_i$$

for

$$d(i) = A_{i-1} \overset{G}{\Rightarrow} S'A_i \overset{G_;}{\Rightarrow} \alpha_i^1 B^1 A'_i \overset{G_;}{\Rightarrow} ... \overset{G_;}{\Rightarrow} \alpha_i^{k(i)-1} B^{k(i)-1} A'_i \overset{G_;}{\Rightarrow} \alpha_i^{k(i)} A'_i.$$

Concatenating all $d'(i)$ in ascending order yields a left-recursive derivation for $w = w_1...w_n$ in $G_;$ (recall: $A_0 = S, A_n = \epsilon$).

□

Proof C.5.5 (Proof of Lemma 7.4.3) We abbreviate $HA_d := domProd(SYN, SEM)$. Given a word $w = w_0 w_1 ... w_n$ such that $w \in L(HA_d)$. Then there is an accepting run $(q_0, q'_0)w_0(q_1, q'_1)...(q_n, q'_n)w_n(q_{n+1}, q'_{n+1})$ of HA_d over w, where $w_i \in symbols^e(HA_d)$ $(1 \leq i \leq n)$.

(1) We show that $q_0 w_0 q_1 ... q_n w_n q_{n+1}$ is an accepting run of SYN over w by showing:

(1.1) $q_0 \in Q^I_{syn}$ and $q_{n+1} \in Q^F_{syn}$.

(1.2) For all $1 \leq i \leq n$ it is true that (q_i, w_i, q_{i+1}) is a valid step in an accepting run of SYN over w according to Defn. 7.4.1.

We conclude:

(1.1) From $(q_0, q'_0) \in Q^I_{HA_d}$ we know that $q_0 \in Q^I_{syn}$. Analogously, $q_{n+1} \in Q^F_{syn}$.

(1.2) Assume $(q_i, q'_i)w_i(q_{i+1}, q'_{i+1}) \in \delta_{HA_d}$. From $w_i \in symbols^e(HA_d)$ we know $w_i \in symbols^e(SYN)$ or $w_i \in \Omega_{sem}$ (recall: SYN may be hierarchical, where SEM has to be basic). We have to distinguish two cases according to Defn. 7.4.4.
Case 1 ($(q_i, a, q_{i+1}) \in \delta_{syn}, a \in HA^s_{syn}, (q'_i, w_i, q'_{i+1}) \in \delta_{sem}, w_i \in L(a)$**)**: Since $w_i \in L(a)$, there is an accepting run of a over w_i. Additionally, $a \in HA^s_{syn}$. Put together, we conclude that (q_i, w_i, q_{i+1}) is a valid step of an accepting run of SYN over w (cf. Defn. 7.4.1).
Case 2 ($(q_i, w_i, q_{i+1}) \in \delta_{syn}, q'_i = q'_{i+1}, w_i \in \Omega_{syn}$**)**: The assumption follows directly (cf. Defn. 7.4.1).

(2) Whenever $w_i \in \Omega_{syn}$, $w_i \notin \Omega_{sem}$ $(\Omega_{syn} \cap \Omega_{sem} = \emptyset)$. Hence, $q'_i = q'_{i+1}$ holds for all transitions $(q_i, q'_i)w_i(q_{i+1}, q'_{i+1})$ in an accepting run over HA_d if $w_i \in \Omega_{SYN}$ (cf. Defn. 7.4.4). Also, $(q'_i, w_i, q'_{i+1}) \in \delta_{sem}$ holds for all transitions $(q_i, q'_i)w_i(q_{i+1}, q'_{i+1})$ if $w_i \in HA^s_{syn}$. Put together, removing steps $q'_i \epsilon q'_i$ from $q'_0 h(w_0) q'_1 ... q'_n h(w_n) q'_{n+1}$ yields an accepting run of SEM over w.

□

Proof C.5.6 (Proof of Lemma 7.4.4) We abbreviate $HA_d := domProd(SYN, SEM)$. Given a state $(q, q') \in Q_{HA_d}$, a word w in $symbols^e(HA)^*$ we prove that HA_d is both, branching deterministic and sequentially deterministic.

(HA_d is branching deterministic)
Given a segmentation $w_1w_2 = w$. Suppose there are transitions $((q, q'), w_1, (q_1, q_1'))$ and $((q, q'), w_1, (q_2, q_2'))$ that can fire on w_1. According to Defn. 7.4.4, both, (q, w_1, q_1) and (q, w_1, q_2) can fire on w_1 in SYN. Since SYN is branching deterministic, $q_1 = q_2$. We remain to show $q_1' = q_2'$. If $w_1 \in \Omega_{syn}$, this directly follows by Defn. 7.4.4 (Dominated Product). If $w_1 \in \Omega_{sem}$, both (q', w_1, q_1') and (q', w_1, w_2') can fire on w_1 in SEM. Since SEM is branching deterministic, $q_1' = q_2'$.

(HA_d is sequentially deterministic)
In any case SYN has to be able to fire on w (Defn. 7.4.4). Hence, there is a unique segmentation $w = w_1w_2$ such that SYN can fire (SYN is sequentially deterministic). If additionally SEM fires, there is another segmentation $w = w_1'w_2'$ such that SEM fires on w_1'. Since both fire, $w_1 = w_1'$ (Defn. 7.4.4). Thus, HA_d is sequentially deterministic.

□

C.6 Proofs for Chap. 11

Proof C.6.1 (Proof of Lemma 11.1.1) We show the assumptions consecutively.

(1) This directly follows from

$$\begin{array}{lll} validBWeb(w) & \Rightarrow \texttt{home}(w) \in \texttt{subDocs}(\texttt{srcDir}(w)) & \text{Def. } validBWeb \text{ (Fig. 11.1)} \\ & \Rightarrow containsDoc(\texttt{srcDir}(w), \texttt{home}(w)) & \text{Def. } containsDoc \text{ (Fig. 11.1)} \\ & \Rightarrow validBWeb(w) & \text{Def. } validAWeb \text{ (Fig. 11.1)} \end{array}$$

(2) We show both directions separately.

($\Longrightarrow$)
We have to show that $validBWeb$ implies the constraints for $BWeb$ in Defn. 11.1.1. We denote them by Defn. 11.1.1(1) to Defn. 11.1.1(4).

(1) We conclude:

$validBWeb(w)$
$\Rightarrow \texttt{home}(w) \in \texttt{subDocs}(\texttt{srcDir}(w)) \wedge \texttt{parseBWeb}(w, \texttt{srcDir}(w), \texttt{name}(w))$ Fig. 11.1
$\Rightarrow \texttt{home}(w) \in \texttt{subDocs}(\texttt{srcDir}(w)) \wedge$
$\quad \texttt{size}(\texttt{subDirs}(\texttt{srcDir}(w))) = 2 \wedge$
$\quad (\exists d' : \texttt{Dir} \bullet d' \in \texttt{subDirs}(\texttt{srcDir}(w)) \wedge \texttt{name}(d') = \text{“}\texttt{html}\text{“}) \wedge$
$\quad (\exists d' : \texttt{Dir} \bullet d' \in \texttt{subDirs}(\texttt{srcDir}(w)) \wedge \texttt{name}(d') = \text{“}\texttt{resources}\text{“}) \wedge$
$\quad \texttt{size}(\texttt{subDocs}(\texttt{srcDir}(w))) = 1 \wedge$
$\quad (\exists d' : \texttt{HTMLDoc} \bullet d' \in \texttt{subDocs}(\texttt{srcDir}(w)) \wedge \texttt{name}(d') = \text{“}\texttt{index.html}\text{“})$ Fig. 11.2
$\Rightarrow$ Defn. 11.1.1(1)

(2) First, the implications

$\texttt{parseTitle}(0, e) = \{\}$
$\Rightarrow (\forall e' : \texttt{HTMLElem} \bullet containsElem(e, e') \Rightarrow \texttt{name}(e') \neq \text{“}\texttt{title}\text{“}) \vee$
$\quad (\exists e' : \texttt{HTMLElem} \bullet$
$\qquad containsElem(e, e') \wedge \texttt{name}(e') = \text{“}\texttt{title}\text{“} \wedge$
$\qquad \forall e'' : \texttt{HTMLTextElem} \bullet (containsElem(e', e'') \Rightarrow \texttt{content}(e'') = \text{“”}))$
$\texttt{parseTitle}(0, e) \neq \{\}$
$\Rightarrow \exists e' : \texttt{HTMLElem} \bullet$
$\qquad containsElem(e, e') \wedge \texttt{name}(e') = \text{“}\texttt{title}\text{“} \wedge$
$\qquad \exists e'' : \texttt{HTMLTextElem} \bullet (containsElem(e', e'') \wedge \texttt{content}(e'') \neq \text{“”})$

hold; this can be seen from the definitions of `parseTitle` and *containsElem*. The proof is omitted for brevity; formally, it goes by induction on the structure of e. We refer to the implications by (*) and (**), respectively, and conclude:

$$
\begin{array}{ll}
& validBWeb(w) \\
\Rightarrow & \texttt{name}(w) = \texttt{name}(\texttt{srcDir}(w)) \wedge \texttt{home}(w) \in \texttt{subDocs}(\texttt{srcDir}(w)) \wedge \\
& \quad \texttt{parseBWeb}(w, \texttt{srcDir}(w), \texttt{name}(w)) & \text{Fig. 11.1} \\
\Rightarrow & \texttt{name}(w) = \texttt{name}(\texttt{srcDir}(w)) \wedge \texttt{home}(w) \in \texttt{subDocs}(\texttt{srcDir}(w)) \wedge \\
& \quad \texttt{size}(\texttt{subDocs}(\texttt{srcDir}(w))) = 1 \wedge \\
& \quad \exists d' : \texttt{HTMLDoc} \bullet \\
& \qquad d' \in \texttt{subDocs}(d) \wedge \texttt{name}(d') = \text{“}\texttt{index.html}\text{“} \wedge \\
& \qquad (\texttt{parseTitle}(0, \texttt{content}(d')) = \{\} \vee \texttt{name}(w) \in \texttt{parseTitle}(0, \texttt{content}(d')) & \text{Fig. 11.2} \\
\Rightarrow & \texttt{name}(w) = \texttt{name}(\texttt{srcDir}(w)) \wedge \\
& \quad (\texttt{parseTitle}(0, \texttt{content}(\texttt{home}(w))) = \{\} \vee \\
& \quad \texttt{name}(w) \in \texttt{parseTitle}(0, \texttt{content}(\texttt{home}(w)))) \\
\Rightarrow & \texttt{name}(w) = \texttt{name}(\texttt{srcDir}(w)) \wedge \\
& \quad ((\forall e' : \texttt{HTMLElem} \bullet \mathit{containsElem}(\texttt{content}(\texttt{home}(w)), e') \Rightarrow \\
& \qquad \texttt{name}(e') \neq \text{“}\texttt{title}\text{“}) \vee \\
& \qquad (\exists e' : \texttt{HTMLElem} \bullet \\
& \qquad\quad \mathit{containsElem}(\texttt{content}(\texttt{home}(w)), e') \wedge \texttt{name}(e') = \text{“}\texttt{title}\text{“} \wedge \\
& \qquad\quad \forall e'' : \texttt{HTMLTextElem} \bullet (\mathit{containsElem}(e', e'') \Rightarrow \texttt{content}(e'') = \text{“”})) \vee & (*) \\
& \qquad (\exists e' : \texttt{HTMLElem} \bullet \\
& \qquad\quad \mathit{containsElem}(\texttt{content}(\texttt{home}(w)), e') \wedge \texttt{name}(e') = \text{“}\texttt{title}\text{“} \wedge \\
& \qquad\quad \exists e'' : \texttt{HTMLTextElem} \bullet \\
& \qquad\qquad \mathit{containsElem}(e', e'') \wedge \texttt{content}(e'') \neq \text{“”} \wedge \texttt{name}(w) = \texttt{content}(e'')) & (**) \\
\Rightarrow & \text{Defn. 11.1.1(2)}
\end{array}
$$

(3) Suppose there is $d : \texttt{Dir}$, $d' : \texttt{HTMLDoc}$, and $w : \texttt{Website}$ such that

$$d \in \texttt{subDirs}(\texttt{srcDir}(w)) \wedge \mathit{containsDoc}(d, d') \wedge \mathit{validBWeb}(w).$$

Due to the definition of `parseBWeb`, this can hold in a directory only if `parseBWeb` succeeds in state one. In $\texttt{subDirs}(\texttt{srcDir}(w))$ this is true for “html” only. Hence, $\texttt{name}(d) = \text{“}\texttt{html}\text{“}$ follows.

(4) Follows similar to (3).

($\Longleftarrow$)

We show

$$\neg \mathit{validBWeb}(w) \;\Rightarrow\; w \text{ does not conform to } \mathit{BWeb}$$

by assuming that at least one of the conjunction terms in the definition of $validBWeb$ does not hold.

Case 1 ($\texttt{name}(w) \neq \texttt{name}(\texttt{srcDir}(w))$):
Trivially, Defn. 11.1.1(2) does not hold.

Case 2 ($\texttt{home}(w) \notin \texttt{subDocs}(\texttt{srcDir}(w))$):
Then Defn. 11.1.1(1) does not hold.

Case 3 ($\texttt{parseBWeb}(0, \texttt{srcDir}(w), \texttt{name}(w)) \neq \texttt{True}$):
If `parseBWeb` does not succeed in states one and two, respectively, Defn. 11.1.1(4) and Defn. 11.1.1(3) do not hold. If it succeeds in both, state one and state two, Defn. 11.1.1(1) or Defn. 11.1.1(2) do not hold.

□

Bibliography

[ABK+02] Egidio Astesiano, Michel Bidoit, Helene Kirchner, Bernd Krieg-Bruckner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: the common algebraic specification language. *Theoretical Computer Science*, 286(2):153–196, September 2002.

[ACM97] Maristella Agosti, Fabio Crestani, and Massimo Melucci. On the use of information retrieval techniques for the automatic construction of hypertext. *Inf. Process. Manage.*, 33(2):133–144, 1997.

[ADK06] Albert Atserias, Anuj Dawar, and Phokion G. Kolaitis. On preservation under homomorphisms and unions of conjunctive queries. *J. ACM*, 53(2):208–237, 2006.

[AFL02] Marcelo Arenas, Wenfei Fan, and Leonid Libkin. What's hard about xml schema constraints? In *DEXA '02: Proceedings of the 13th International Conference on Database and Expert Systems Applications*, pages 269–278, London, UK, 2002. Springer-Verlag.

[AKY99] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating hierarchical state machines. In *Automata, Languages and Programming: 26th International Colloquium, ICALP'99*, pages 703–703, Prague, Czech Republic, July 1999. Springer (LNCS Vol. 1644). `www.springerlink.com/content/980rt2de1mqqu2ll`.

[BALD03] Béatrice Bouchou, Mirian Halfeld Ferrari Alves, Dominique Laurent, and Denio Duarte. Extending tree automata to model xml validation under element and attribute constraints. In *ICEIS*, volume 1, pages 184–190, 2003.

[Bar04] Erik Peter Barnsleben. Database migration: A literature review and case study, November 2004. Master's thesis.

[BFM05] Denilson Barbosa, Juliana Freire, and Alberto O. Mendelzon. Designing information-preserving mapping schemes for xml. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 109–120. VLDB Endowment, 2005.

[BM04] Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[Bor07] Daniel Borkowitz. Konzeption, Spezifikation und Realisierung einer Transformation von Kernelementen der XHTML- und CSS-Spezifikation in das Open Document Format (ODF), 2007. Master's thesis, Universität der Bundeswehr München, UniBwM-ID 22/2007.

[BRSS06] Uwe M Borghoff, Peter Rödig, Jan Scheffczyk, and Lothar Schmitz. *Long-Term Preservation of Digital Documents*. Springer Verlag, Heidelberg, 2006.

[BS03] Egon Börger and Robert Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer Verlag, Heidelberg, 2003.

[BZ00] Hubert Baumeister and Alexandre V. Zamulin. State-based extensions of CASL. In *Integrated Formal Methods (IFM)*, pages 3–24. Springer, 2000.

[CDG+07] Hubert Comon, Max Dauchet, Remi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sohie Tison, and Marc Tommasi. Tree automata techniques and applications, October 2007.
`www.grappa.univ-lille3.fr/tata`.

[Chi00] Boris Chidlovskii. Using regular tree automata as XML schemas. In *Proc. Int. IEEE Conf. on Advances in Dig. Lib (ADL2000)*, pages 89–98, Washington, DC, USA, May 2000.

[CLB01] James Cheney, Carl Lagoze, and Peter Botticelli. Towards a theory of information preservation. Technical report, Cornell University, Ithaca, New York, 2001.

[CoF04] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[Con02] Consultative Committee for Space Data Systems. Reference model for an open archival information system. Technical report, Space Data Systems, January 2002.
`public.ccsds.org/publications/archive/650x0b1.pdf`.

[DCMID08] The Dublic Core Metadata Initiative (DCMI). DCMI metadata terms, January 2008.
`dublincore.org/documents/dcmi-terms/`.

[de 86] D. de Champeaux. Subproblem finder and instance checker, two cooperating modules for theorem provers. *Journal of the ACM*, 33(4):633–657, 1986.

[Dig01] Digital Preservation Testbed. Migration: Context and current status, 2001. White Paper,
`www.digitaleduurzaamheid.nl/bibliotheek/docs/Migration.pdf`.

[EGdL+05] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 2005.

[EGL89] Hans-Dieter Ehrich, Martin Gogolla, and Udo W. Lipeck. *Algebraische Spezifikationen abstrakter Datentypen.* B. G. Teubner, Stuttgart, 1989. ISBN 3-519-02266-4.

[Erw96] Martin Erwig. Active patterns. In *Proc. of the 16th Int. Workshop on Implementation of Functional Languages*, volume 1268, pages 21–40, Bonn-Bad-Godesberg, GE, Sep. 1996. Springer LNCS.

[ES07] Jérôme Euzenat and Pavel Shvaiko. *Ontology matching.* Springer-Verlag, Heidelberg (DE), 2007.

[FBR06] Miguel Ferreira, Ana Alice Baptista, and José Carlos Ramalho. A foundation for automatic digital preservation. *Ariadne*, 48, July 2006.

[Fla04] Stephan Flake. Towards the completion of the formal semantics of ocl 2.0. In *ACSC '04: Proceedings of the 27th Australasian conference on Computer science*, pages 73–82, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[FM02] Stephan Flake and Wolfgang Mueller. An OCL extension for real-time constraints. In *Object Modeling with the OCL*, Berlin / Heidelberg, 2002. Springer.

[FP00] Piero Fraternali and Paolo Paolini. Model-driven development of web applications: the AutoWeb system. *ACM Trans. Inf. Syst.*, 18(4):323–382, 2000.

[GF02] Marcos André Gonçalves and Edward A. Fox. 5SL - a language for declarative specification and generation of digital labraries. In *Joint Conference on Digital Libraries 2002 (JCDL'02)*. ACM, June 2002.

[GFWK04] Marcos André Gonçalves, Deward A. Fox, Layne T. Watson, and Neill A. Kipp. Streams, structures, spaces, scenarios, societies (5S): A formal model for digital libraries. *ACM Transactions on Information Systems*, 22(2):270–312, April 2004.

[Gol97] Gene Golovchinsky. What the query told the link: the integration of hypertext and information retrieval. In *HYPERTEXT '97: Proceedings of the eighth ACM conference on Hypertext*, pages 67–74, New York, NY, USA, 1997. ACM.

[Goo07] The google library program, 2007. `books.google.com/googlebooks/library.html`.

[GPN96] Pedro Palao Gostanza, Ricardo Pena, and Manuel Núnez. A new look at pattern matching in abstract data types. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 110–121, New York, NY, USA, 1996. ACM.

[GRB03] Martin Gogolla, Mark Richters, and Jörn Bohling. Tool support for validating uml and ocl models through automatic snapshot generation. In *Proceedings of theAnnual Research Conference South African Institute of Computer Scientists and Information Technologists on Enablement through Technology (SAICSIT'2003)*, pages 248–257, 2003.

[Grz97] Grzegorz Rozenberg et al. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific Publishing, New Jersey, 1997.

[GS97] Yuri Gurevich and Marc Spielmann. Recursive abstract state machines. *J.UCS: Journal of Universal Computer Science*, 3(4):233–246, 1997.

[Gur00] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, 2000.

[GW05] David Giaretta and Heather Weaver. Digital curation and preservation: Defining the research agenda for the next decade. Technical report, Digital Curation Centre, November 2005.
`www.dcc.ac.uk/events/warwick_2005/Warwick_Workshop_report.pdf`.

[GYS07] Fausto Giunchiglia, Mikalai Yatskevich, and Pavel Shvaiko. Semantic matching: Algorithms and implementation. *Journal on Data Semantics IX*, pages 1–38, 2007.

[Hai05] Jean L. Hainaut. Transformation-based database engineering. In L. Rivero, J. Doorn, and V. Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*. IDEA Group, 2005.

[HC04] J. Hunter and S. Choudhury. A semi-automated digital preservation system based on semantic web services. *Digital Libraries, 2004. Proceedings of the 2004 Joint ACM/IEEE Conference on Digital libraries*, pages 269–278, 7-11 June 2004.

[HD04] Johannes Henkel and Amer Diwan. A tool for writing and debugging algebraic specifications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 449–458. IEEE Computer Society, 2004.

[HKKR05] Martin Hitz, Gerti Kappel, Elisabetz Kapsammer, and Werner Retschitzegger. *UML@Work*. dpunkt Verlag, Heidelberg, 3rd edition, 2005.

[HL07] Michael Huggett and Joel Lanir. Static reformulation: a user study of static hypertext for query-based reformulation. In *JCDL '07: Proceedings of the 7th ACM/IEEE joint conference on Digital libraries*, pages 319–328, New York, NY, USA, 2007. ACM.

[HLSU02] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 327–341, London, UK, 2002. Springer-Verlag.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 3rd edition, July 2006. ISBN: 978-0321455369.

[Int98] Internet Engineering Task Force (IETF), Network Working Group. Uniform resource identifiers (URI): Generic syntax, 1998. RFC2396, `www.ietf.org/rfc/rfc2396.txt`.

[Isa02] *Isabelle/HOL: a proof assistant for higher-order logic.* Springer-Verlag, London, UK, 2002.

[Jon95] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, pages 97–136, 1995.

[KK96] Hans-Jörg Kreowski and Sabine Kuske. On the Interleaving Semantics of Transformation Units—A Step into GRACE. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073, pages 89–106. Springer-Verlag, 1996.

[KS03] Yannis Kalfoglou and Marco Schorlemmer. If-map: An ontology-mapping method based on information-flow theory. *Journal on Data Semantics*, pages 98–127, 2003.

[LS06] Monika Lanzenberger and Jennifer Sampson. Alviz - a tool for visual ontology alignment. In *IV '06: Proceedings of the conference on Information Visualization*, pages 430–440, Washington, DC, USA, 2006. IEEE Computer Society.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Mai06] Jacques le Maitre. Describing multistructured XML documents by means of delay nodes. In *Proc. of the ACM Symp. on Doc. Eng. (DocEng 2006)*, pages 155–164, Amsterdam, The Netherlands, October 2006.

[MBDH02] Jayant Madhavan, Philip A. Bernstein, Pedro Domingos, and Alon Y. Halevy. Representing and reasoning about mappings between domain models. In *Eighteenth national conference on Artificial intelligence*, pages 80–86. American Association for Artificial Intelligence, 2002.

[McC03] William McCune. Mace4 reference manual and guide. Technical report, Argonne National Laboratory, Mathematics and Computer Science Division, 2003. Technical Memo ANL/MCS-TM-264.

[McC06] William McCune. Prover9 manual. Technical report, Argonne National Laboratory, Mathematics and Computer Science Division, 2006.

[MCG05] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. 04101 discussion – a taxonomy of model transformations. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[MDJ02] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation, volume 2505 of Lecture Notes in Computer Science*, pages 286–301. Springer, 2002.

[MDM+94] A. Meier, R. Dippold, J. Mercerat, A. Muriset, J.-C. Untersinger, R. Eckerlin, and F. Ferrara. Hierarchical to relational database migration. *Software, IEEE*, 11(3):21–27, May 1994.

[MEDJ05] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations: Research articles. *J. Softw. Maint. Evol.*, 17(4):247–276, 2005.

[MLS97] Erich Mikk, Yassine Lakhnech, and Michael Siegel. Hierarchical automata as model for statecharts. In *Advances in Computing Science - ASIAN'97*, pages 181–196. Springer (LNCS Vol. 1345), 1997.
`www.springerlink.com/content/vk4328r678n1k036`.

[MOSS99] Markus Müller-Olm, David A. Schmidt, and Bernhard Steffen. Model-checking: A tutorial introduction. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, pages 330–354, London, UK, 1999. Springer-Verlag.

[MP99] Peter McBrien and Alexandra Poulovassilis. A uniform approach to inter-model transformations. In *Proceedings of the 11th International Conference on Advanced Information Systems Engineering (CAiSE'99)*, volume 1626, pages 333–348. Springer Verlag LNCS, 1999.

[NCEF02] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Trans. Inter. Tech.*, 2(2):151–185, 2002.

[Nev02] Frank Neven. Automata, logic, and XML. In *Proc. of the 16th Int. Workshop CSL 2002, 11th Ann. Conf. of the EACSL*, volume 2471, pages 2–26, Edingburgh, UK, Sep. 2002. Springer LNCS.

[Obj05] Object Management Group. QVT: Revised submission for MOF 2.0 Query/View/Transformation RFP. Request for Proposal, 2005. ad/2002-04-10 version 2.1.

[OC03] Christopher Olston and Ed H. Chi. Scenttrails: Integrating browsing and searching on the web. *ACM Trans. Comput.-Hum. Interact.*, 10(3):177–197, 2003.

[Ock98] John Ockerbloom. Mediating among diverse data formats. Technical report, Carnegie Mellon Computer Science, 1998.

[Org06] Organization for the Advancement of Structured Information Standards (OASIS). Open Document Format for Office Applications (OpenDocument) v1.0 (Second Edition), 2006. ISO Standard ISO/IEC 26300:2006, `www.oasis-open.org/committees/download.php/19274/OpenDocument-v1.0ed2-cs1.pdf`.

[PHS+06] Stoyan Paunov, James Hill, Douglas Schmidt, Steven D. Baker, and John M. Slaby. Domain-specific modeling languages for configuring and evaluating enterprise DRE system quality of service. In *Proc. 13th IEEE Int. Symp. on Eng. of Comp. Based Sys. (ECBS'06)*, pages 196–208, Washington, DC, USA, 2006. IEEE Computer Society.

[Por05] Ivan Porres. Rule-based update transformations and their application to model refactorings. *Software and System Modeling*, 4(4):368–385, 2005.

[PRE05] PREMIS working group. Data dictionary for preservation metadata. Technical report, OCLC: Dublin, Ohio, RLG: Mountain View, California, 2005. `www.oclc.org/research/projects/pmwg/premis-final.pdf`.

[Rei03a] Wolfgang Reisig. On Gurevich's Theorem on Sequential Algorithms. *Acta Informatica*, 39(5):273–305, 2003.

[Rei03b] Wolfgang Reisig. The Expressive Power of Abstract State Machines. *Computing and Informatics*, 22(3):209–219, 2003.

[Ric02] Mark Richters. A precise approach to validating UML models and OCL constraints. In Martin Gogolla, Hans-Jörg Kreowski, Bernd Krieg-Brückner, Jan Peleska, and Bernd-Holger Schlingloff, editors, *BISS Monographs*, volume 14. Logos Verlag, Berlin, 2002. ISBN: 3-98722-842-4.

[Rot99] Jeff Rothenberg. *Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation.* Council on Library & Information Resources, 1999.

[RS01] Markus Roggenbach and Lutz Schröder. Towards trustworthy specifications I: Consistency checks. *Lecture Notes in Computer Science*, 2267:305+, 2001.

[RSV04] A. Rensink, Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In H. Ehrig, G. Engels, F. Parise-Presicce, and G. Rozenberg, editors, *International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 226–241, Berlin, 2004. Springer Verlag.

[SBNR07] Stephan Strodl, Christoph Becker, Robert Neumayer, and Andreas Rauber. How to choose a digital preservation strategy: evaluating a preservation planning procedure. In *JCDL '07: Proceedings of the 2007 conference on Digital libraries*, pages 29–38, New York, NY, USA, 2007. ACM Press.

[SBRS04] Jan Scheffczyk, Uwe M. Borghoff, Peter Rödig, and Lothar Schmitz. Managing inconsistent repositories via prioritized repair actions. In *Proc. of the ACM Symp. on Doc. Eng. (DocEng 2004)*, pages 137–146, 2004.

[Sch04] Jan Scheffczyk. *Consistent Document Engineering.* PhD thesis, Universität der Bundeswehr München, Fakultät für Informatik, Neubiberg, August 2004.

[SKB06] T. Szemethy, G. Karsai, and D. Balasubramanian. Model transformations in the model-based development of real-time systems. In *Proc. 13th IEEE Int. Symp. on Eng. of Comp. Based Sys. (ECBS'06)*, pages 188–196, Washington, DC, USA, 2006. IEEE Computer Society.

[SL05] Andreas U. Schmidt and Zbyněk Loebl. Legal security for transformations of signed documents: Fundamental concepts. In *Proceedings of the 2nd European PKI Workshop*, volume 3545, pages 255–270. Springer Verlag (LNCS), June 2005.

[SM01] Gerd Stumme and Alexander Maedche. Ontology merging for federated ontologies on the semantic web. In *Proc. Int. Workshop for Foundations of Models for Information Integration (FMII-2001)*, 2001.

[SRR+06] Stephan Strodl, Andreas Rauber, Carl Rauch, Hans Hofman, Franca Debole, and Giuseppe Amato. The DELOS testbed for choosing a digital preservation strategy. In *Digital Libraries: Achievements, Challenges and Opportunities*, volume 4312, pages 323–332. Springer Verlag (LNCS), November 2006.

[SSR94] Edward Sciore, Michael Siegel, and Arnon Rosenthal. Using semantic values to facilitate interoperability among heterogeneous information systems. *ACM Trans. Database Syst.*, 19(2):254–290, 1994.

[SWZ99] Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: language and environment. In *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*, pages 487–550. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[Tal98] Carolyn Talcott. Reasoning about functions with effects. In *Higher order operational techniques in semantics*, pages 347–390, New York, NY, USA, 1998. Cambridge University Press. ISBN: 0-521-63168-8.

[Tas96] Task Force on Archiving of Digital Information. Preserving digital information. Technical report, Research Library Ground (RLG), May 1996. `www.rlg.org/legacy/ftpd/pub/archtf/final-report.pdf`.

[TB06] Thomas Triebsees and Uwe M. Borghoff. Preservation-centric and constraint-based migration of digital documents. In *DocEng '06: Proceedings of the 2006 ACM Symposium on Document Engineering*, pages 59–61, New York, NY, USA, 2006. ACM Press.

[TB07a] Thomas Triebsees and Uwe M. Borghoff. A theory for model-based transformation applied to computer-supported preservation in digital archives. In *Proc. 14th Ann. IEEE International Conference on the Engineering of Compupter Based Systems (ECBS'07)*, Tucson, AZ, USA, March 2007. IEEE Computer Society Press.

[TB07b] Thomas Triebsees and Uwe M. Borghoff. Towards automatic document migration: semantic preservation of embedded queries. In *DocEng '07: Proceedings of the 2007 ACM symposium on Document engineering*, pages 209–218, New York, NY, USA, 2007. ACM Press.

[TB07c] Thomas Triebsees and Uwe M. Borghoff. Towards constraint-based preservation in systems specification. In *Proceedings of the International Conference on Conputer-aided Systems Theory (EuroCAST07)*, volume 4739, pages 894–902. Springer Verlag (LNCS), February 2007.

[TBS05] Thomas Triebsees, Uwe M. Borghoff, and Jan Scheffczyk. Controlled migration in digital archives. In *Proceedings of the International Conference on Digital Archive Technologies (ICDAT2005)*, pages 5–19, June 2005.

[Tho99] Simon Thompson. *The Craft of Functional Programming.* Addison-Wesley Longman, Amsterdam, 2nd edition, 1999. ISBN 978-0201342758.

[Ton97] Hans Tonino. *A Theory of Many-sorted Evolving Algebras.* PhD thesis, Delft University of Technology, Delft, 1997.

[Ton98] Hans Tonino. A sound and complete SOS-semantics for non-distributed deterministic abstract state machines. In *Workshop on Abstract State Machines*, pages 91–110, 1998.

[Top01] TopicMaps.Org Authoring Group. XML Topic Maps (XMT) 1.0, June 2001. `www.topicmaps.org/xtm/index.html`.

[Tri07] Thomas Triebsees. Constraint-based model transformation: Tracing the preservation of semantic properties. *Journal of Software*, 2(2):1–11, 2007.

[Val04] Valoris. Comparative assessment of open documents formats market overview. Technical report, European Commission, Specific agreement no.3-IDA.20030523, 2004.

[Wan60] Hao Wang. Toward mechanical mathematics. *IBM Journal of Research and Development*, 4(1):2–22, 1960.

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, New Jersey, 1996.

[Whe01] Paul Wheatley. Migration — a CAMiLEON discussion paper, 2001. `www.ariadne.ac.uk/issue29/camileon/intro.html`.

[Wir95] Martin Wirsing. Algebraic specification languages: An overview. In *Recent Trends in Data Type Specification*. Springer, Berlin / Heidelberg, 1995.

[WO00] Jeanette M. Wing and John Ockerbloom. Respectful type converters. *IEEE Trans. on Software Eng.*, 26:579–593, July 2000.

[Wor99] World Wide Web Consortium W3C. RDF primer, 1999. W3C Recommendation, `www.w3.org/TR/REC-rdf-syntax/`.

[Wor01] World Wide Web Consortium W3C. XML Schema part 0: Primer, 2001. W3C Recommendation, `www.w3.org/TR/xmlschema-0/`.

[Wor02] World Wide Web Consortium W3C. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition), 2002. W3C Recommendation, `www.w3.org/TR/xhtml1/`.

[Wor03] World Wide Web Consortium W3C. XML path language (XPath) 2.0, 2003. W3C Recommendation, `www.w3.org/TR/xpath20/`.

[Wor04a] World Wide Web Consortium W3C. Document Object Model (DOM) Level 3 Core Specification, 2004. W3C Recommendation, `www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`.

[Wor04b] World Wide Web Consortium W3C. Extensible Markup Language (XML) 1.1, 2004. W3C Recommendation, `www.w3.org/TR/xml11`.

[Wor07a] World Wide Web Consortium W3C. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, 2007. W3C Candidate Recommendation, `www.w3.org/TR/CSS21/`.

[Wor07b] World Wide Web Consortium W3C. XQuery 1.0: An XML Query Language, 2007. W3C Recommendation, `www.w3.org/TR/xquery/`.

[WS99] Ross Wilkinson and Alan F. Smeaton. Automatic link generation. *ACM Computing Surveys*, pages 27–40, December 1999.

[Zam97] Alexandre Zamulin. Typed gurevich machines revisited. Joint CS & IIS Bulletin, Computer Science, 1997.

[Zam98] Alexandre Zamulin. Specification of Dynamic Systems by Typed Gurevich Machines. In Z. Bubnicki and A. Grzech, editors, *Proceedings of the 13th International Conference on System Science*, pages 160–167, Wrocłlaw, Poland, 15-18 1998.

[ZG03] Paul Ziemann and Martin Gogolla. OCL extended with temporal logic. In *Perspectives of System Informatics: 5th International Andrei Ershov Memorial Conference*, Berlin / Heidelberg, 2003. Springer.