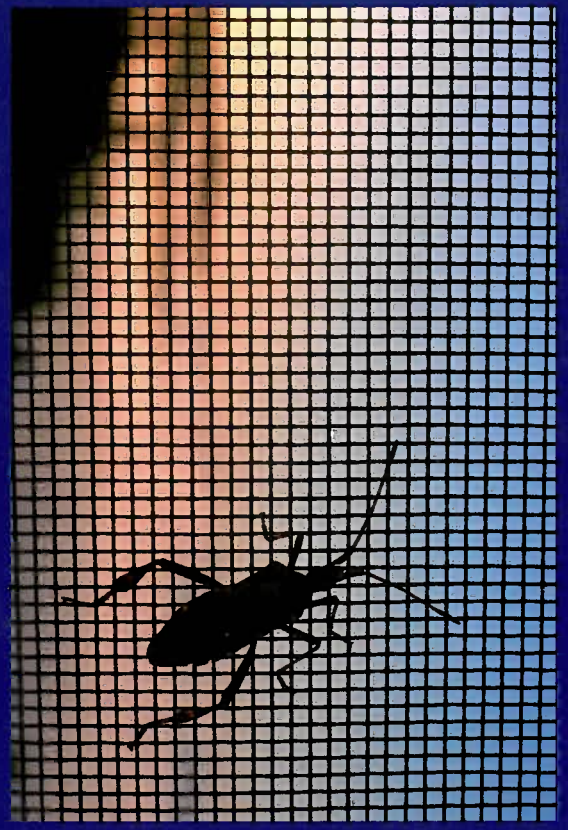

Jens R. Calamé

Testing Reactive Systems with Data

Enumerative Methods and Constraint Solving

Testing Reactive Systems with Data

Enumerative Methods and Constraint Solving

Jens R. Calamé

All web addresses, which this thesis refers to, have last been visited on 26-03-2008.

Typeset by Jens R. Calamé with LaTeX 2ε using the Palatino, 10pt

Cover: Idea and realization Jens R. Calamé
Image "Silhouette of Bug on Screen" © Barbara Chase/Corbis

Printed by Ponsen & Looijen B.V., Wageningen

ISBN 978-90-6464-273-9
IPA dissertation series 2008-20

The work reported in this thesis has been carried out at the CWI (Centrum Wiskunde & Informatica) within the TT-Medal project (Test & Testing Methodologies for Advanced Languages) and the BRICKS project funded by the *Besluit Subsidies Investeringen Kennisinfrastructuur* (BSIK). The research took place under the auspices of the *Instituut voor Programmatuurkunde en Algoritmiek* (IPA).

TESTING REACTIVE SYSTEMS WITH DATA

ENUMERATIVE METHODS AND CONSTRAINT SOLVING

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente
op gezag van de rector magnificus,
prof. dr. W.H.M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 4 september 2008 om 13:15 uur

door

Jens Rüdiger Calamé

geboren op 21 juni 1979
te Hamburg, Duitsland

Dit proefschrift is goedgekeurd door de promotoren:

prof. dr. J.C. van de Pol
prof. dr. W.J. Fokkink

Contents

Preface

About ten years have now passed by since my focus has for the first time been turned to software quality assurance by a book named *No More Bugs* (Maguire, 1998) – and a few more years probably since I have for the first time successfully implemented a bug myself. Now, after having developed loads of more bugs and after having worked for four years by now on the question how to sort them out again, you hold my small contribution to the area of software quality assurance in your hands.

When I write *my* contribution, I have to be evocative of all the people who made this book possible. First of all, I would like to thank my promotors Jaco van de Pol and Wan Fokkink. Jaco's sharp eyes and his constructive critics over all the years have been substantial ingredients in producing a thesis with a minimal amount of bugs in it. Wan contributed a lot of very constructive comments on the drafts of this book. It was also him who made it possible for me to work at CWI in the first place.

Next, I would like to thank the members of the reading committee of my thesis, in alphabetical order: Mehmet Aksit, Thomas Arts, Ed Brinksma, Ina Schieferdecker, Natalia Sidorova, and Mariëlle Stoelinga. Their feedback was an indispensable contribution to the quality of this thesis. I would also like to thank Ina for laying the contact with Wan in 2004. Furthermore, I would like to thank Natalia for her input on the papers this thesis is based upon, especially on that on *Bug Hunting with False Negatives*.

At CWI I had the opportunity to work in a vivid working group with very nice colleagues. I would especially like to thank Natalia Yustinova for more than two years of productive cooperation. I really appreciated the inspiring discussions with you! Also your comments on the introduction of this thesis have turned out very helpful to me. Furthermore, I would like to thank Bert Lisser for his support guiding me through the shallows of μCRL and its associated libraries. Finally, I will keep Anton Wijs in mind as a very nice room mate, and Mohammad Torabi Dashti, Yanjing Wang and Taolue Chen as colleagues with whom I could also have a great time at CWI and elsewhere.

Finishing up the thesis, I would like to name a few more people whose advice I appreciated in these last months. First of all, I would like to thank Marco Kuhrmann for meeting the challenge to proof-read this book and for providing a number of useful comments on its contents. Having a look in the back of the book, I would like to thank Wan, Anton, Stefan Blom and Jaco for hunting bugs in my Dutch summary. And having a look at the book's appearance, I would like to thank Manuel Voigt for helping me finding an appropriate cover image, and Tobias Baanders for his kind advice on improvements regarding the cover as such. Finally, I would also like to thank Paul Klint for supporting me in organizational issues in my last months at CWI.

Doing research and writing a thesis is – in most cases at least – embedded into something called *life*. Last but not least, I would like to turn the attention to this aspect. As a start, I would like to thank my landlady Mieke Langereis for more than 2 years of very kind hosting, which included not only a roof above my head, but also warmth, tasty food, lots of tea and long conversations in which I could improve my Dutch a lot. I also really enjoyed mutual visits with my friends and hours with them on the phone or in Internet chats. Especially in the last time of my studies, the Internet turned out to be a very important utility to keep my social life up and running. At last, I would like to thank my family for all their support and their imperturbable belief in the decisions I have made in my life so far.

My belief in finally overcoming the problem of faulty software is not as imperturbable. I actually do not know, whether we will ever reach the state of no more bugs in software engineering. Already the Romans knew that *errare humanum est* (for the main audience of this book: human $\rightarrow \Box\Diamond$ error), so I personally doubt bug-freeness as long as computers are programmed with human beings involved. However, we can try to repress bugs as much as possible, and I hope that this thesis is a bit of a help on the long way doing so.

東京, Japan, June 2008

Jens R. Calamé

Chapter 1

Introduction

> Testing is a very inefficient way of convincing oneself of the correctness of a program.
>
> *(Edsger W. Dijkstra[1])*

Quality is defined as the *"degree to which a set of inherent characteristics fulfills requirements"* (ISO 9000:2000; cf. Hoyle, 2005). Measuring – and consequently also ensuring – quality is a necessity in the development of any kind of product in order to unveil imperfections, which can lead to more or less severe problems in handling the particular product. Every class of products has thereby its own set of characteristics to be measured. For software, those are defined by ISO 9126 as *functionality*, *reliability*, *usability*, *efficiency*, *maintainability* and *portability* with several subcharacteristics (Hendriks et al., 2002).

There exist several approaches to measure the quality of a software product. Each of them focuses on a subset of the above-mentioned characteristics. Amongst other things, the approaches differ in the analyzed artifacts, like a model of the software product (*model checking*), its source code (*static analysis* or *code reviews*), the product's behavior (*software tests*) or even the development process of the product (*appraisal methods*). The outcome of such measurements can in many cases be directly interpreted, if, for instance, a model has been considered incorrect by model checking. In other cases, the outcome serves as an input to metrics (cf. Kan, 2003), which then support the improvement of the product and eventually even that of its successors.

Assuming a certain maturity, the development of a software product follows a software development process. Such a process defines development steps in project stages. Amongst other things, these stages contain the definition of requirements for the developed software, the design of the software system and its components, as well as their realization. Measuring software quality is a continuous activity, accompanying the whole software development process.

In this work, we will concentrate on two of the named activities supporting quality measurement, *software test* and *model checking*. The one we will mainly focus on, is the software test. As it has already been hinted, this is a dynamic approach, which runs a series of experiments on the software product in order to derive, whether it matches the customer's expectations. The second approach considered in this work is model checking, a representative for formal quality measurement. It formally verifies whether the software model meets a particular requirement, rather than making an experiment on the software, as does testing.

[1] at the NATO Software Engineering Conference 1968

In the remainder of this chapter, we will first introduce both approaches, then briefly present the projects, in which this work was carried out, and finally discuss the contributions and the structure of this thesis.

1.1 Software Testing

The software test is a widely-accepted dynamic approach to measure the quality of a software product. During a test, the use of this product is simulated in a certain scenario. Within a software development process, testing often happens at a relatively late stage, since the product under test or at least some artifacts of this product must be available. However, test cases, i.e. the scenarios under which the product is tested, can already be developed in parallel to the product itself.

Even though, the test in most cases appears as only one or two (atomic) steps in common software development processes, it is a process on its own. There are several standards for test processes (e.g. BS7925-2 and IEEE 1008 for the test of single components, as well as, for instance, *TMap* as an industrial approach to testing; cf. Van Veenendaal, 2002). It should be sufficient for our work, that these processes have a common structure, namely the steps *planning*, *specification* and *execution* of the test as well as an adjacent interpretation of its results.

1.1.1 A Short History of Software Testing

Testing in computer science has a long history, which goes back to the days of the first computing machines, and has since then changed its objective several times. We will here give a brief overview of this historic development, to clarify today's comprehension of the meaning of software testing in software development.

Gelperin and Hetzel (1988) have identified five main periods in the history of software testing. In the first, *debugging-oriented*, period, the two activities *testing* and *debugging*, which later got different meanings, were intertwined and could even be used synonymously, since they both targeted the elimination of programming faults.

This period was replaced around 1957 by the so-called *demonstration-oriented* period. Here, we encounter a strict distinction of debugging as the activity to *"make sure, the program runs"* and testing as the activity to *"make sure, the program solves the problem"* (Gelperin and Hetzel, 1988). This means in short, that testing was used to show that the program under investigation was *correct*.

With this idea in mind, the world stumbled into the software crisis of 1968 (Naur and Randell, 1969). For the area of testing, it then took about one more decade, until Myers (1979) made the step from testing as a correctness proof to testing as *fault detection*. Debugging was from then on defined as the activity to *fix faults*.

Gelperin and Hetzel name this period from 1979 on *destruction-oriented*. It was followed by two more periods in which, according to the authors, testing was first integrated into a unified product evaluation process, and then its focus was shifted

from just finding existing faults to also preventing the occurrence of new ones. Accordingly, over the years the position of testing in software development processes gradually shifted from a fixed activity at the end (like in the *waterfall model*) to one which continuously accompanies the implementation of the software product itself like in the *Rational Unified Process* (cf. Kruchten, 2003).

In the twenty years after the publication of the article, finally, the flavor of testing has surely changed further, for instance with the advent of test-first approaches in Extreme Programming (Beck, 1999), where test cases are even used as software specifications. The main focus of testing as fault detection, however, has stayed unchanged for already nearly 30 years by now.

What are faults, bugs and – failures?

In the context of software quality, three terms are regularly used to point out problems: faults, bugs and failures. The term *fault* or *bug* is defined by the ISTQB (2006) as a flaw, which makes a system deviate from its required functionality. Following this definition, a *failure*, i.e. the actual visibly wrong behavior, is caused by a fault. In the remainder of this thesis, we will mainly use the term *failure* in the context of testing. The term *bug* will later be used in the context of model checking.

1.1.2 Testing and Formal Methods

Testing is in many cases performed in an adhoc manner. This means, that tests are developed on the basis of informal system requirements and specifications and then – sometimes still manually – executed. Testing, however, is tedious and costly, so that attempts are made to automate it.

In particular, two aspects of testing can be automated: test generation and test execution. The automation of *test execution* can be already achieved with rather simple tools, like a manually implemented test driver for a test on the level of the Application Programming Interface (API), using, for instance, Unit Testing Frameworks, or a capture-and-replay tool for the test of a graphical user interface. The tools themselves do not have to *understand*, what they are doing, they just have to repeat a task over and over again.

Things are different when it comes to the automation of *test generation*. In this case, tools have to develop a certain understanding of the entities on which they are acting. Informal system requirements and specifications cannot be interpreted automatically, so that those artifacts must be *formalized*. Formal methods are based on mathematical notions for the specification of requirements and specifications. Since these constructs are inherently well-defined, it is possible for automatic tools to interpret them in a non-ambiguous manner.

1.2 Model Checking

1.2.1 A Short Introduction to Model Checking

Model checking (Clarke et al., 1999; Bérard et al., 2001) is a formal method to verify the correctness of a software model. The verification process is performed automatically by exhaustively searching all states of the system under investigation for possible faults. The absence of faults may then be interpreted as a sign that the model is indeed correct.

In order to model check a system, the system must be specified. A system specification on which model checking can be applied, must be formal. Such a formal specification can, for instance, be an automaton, a Petri net or a process-algebraic specification (Petri, 1962; Bergstra et al., 2001). The requirements which the system is supposed to fulfill, must also be defined. In most cases, these requirements are given as formulae in a temporal logic, i.e. a logic, which can assert the evolution of the system over time.

In the actual verification, the model checker takes both the system specification and the requirement as an input and does an exhaustive search over the system's state space. If it does not find a bug, the system can safely be considered *correct* w.r.t. the requirement. If it finds a bug, the system is not correct and a counterexample is returned. This counterexample shows under which circumstances the found bug can be reproduced.

One approach in model checking is *enumerative* model checking, which expands all occurrences of data variables to all possible values for these variables, and examines the resulting state space. The search over the state space of the system is an exhaustive one. This can lead to problems with systems, which have a large or even infinite state space – and indeed, it does with most systems. This critical phenomenon is known as *state space explosion*, and in most literature refered to as *state explosion*.

In order to understand this problem, we have to introduce the notion of states. In the course of its execution, a system follows a particular trace of instructions, the so-called *control flow*[2]. If the system is data-dependent, the instructions in the control flow interact with and change data values. The subsequent interaction with data leads to a system's *data flow*. Finally, a state of a system is the product of its position in its control flow together with the values of all data items in the system.

To make it short: The more possible values can be selected for a data item, i.e. the bigger the data item's domain is, the more states a system can have – only regarding this one data item and not even the other ones. This means that the state space of a system grows over all limits by enumerating over large or infinite data domains. Model checking as an exhaustive search in a state space has to address this problem.

This can be achieved in two ways. The first one is to use abstraction techniques, the number of data values and hence the number of states in the system can often be reduced, so that enumerative model checking can still be applied.

[2]To clarify the exposition, we do not consider parallel processes here.

The second way to address the problem of state space explosion is to not enumerate over all data values in the system at all, but to regard the variables themselves instead during the whole verification process. This approach is named *symbolic* model checking[3].

1.2.2 Software Testing and Model Checking

Software testing and model checking are two orthogonal aspects of software quality assurance. While the first approach dynamically simulates the usage of the implementation, i.e. the final product, the second one statically verifies an intermediate product of the software development process, the model. This intermediate product is not as complete as the final one, since a model is always only an image of reality (lat. *modellus*: at a small scale). This is, however, an advantage of model checking with respect to testing, since a model can more easily be checked *completely* (at least regarding the checked system properties), while completeness in testing is hardly ever achieved. Since the two approaches are orthogonal, they complement each other: Performing model checking at a relatively early stage of the software development process already allows to reduce the number of bugs before they are actually implemented. This decreases the number of testing and debugging cycles, and moreover diminishes costs for a redesign of the whole application in its implementation phase.

1.3 Structure and Contribution of this Thesis

Much of the work described in this thesis was done in the scope of the European project *Tests & Testing Methodologies for Advanced Languages* (TT-Medal)[4]. Its objective was the development of methodologies, tools and industrial experience to improve the effectiveness and efficiency of testing for the European industry (see TT-Medal, online). In order to achieve this objective, methodologies have been developed for the application of the Testing and Test Control Notation, version 3 (TTCN-3) outside of its home domain telecommunication. TTCN-3 is a language to define modular platform-independent test cases, which are subsequently executed by the TTCN-3 runtime environment. Those methodologies developed in TT-Medal were to be based on a generic testing infrastructure. The domains to which TTCN-3 was applied were the automotive sector, finances and railways. The research topics touched in the project comprised the generation of test cases in TTCN-3 especially regarding the aspect of test coverage, the validation of test cases, regression test techniques for system integration and the reuse of test cases between phases of a test process as well as software products.

[3] "Symbolic" in this context should not be confused with approaches to model checking, which make use of BDDs in order to store sets of states and to which the term "symbolic" model checking normally refers.

[4] TT-Medal won the ITEA Achievement Award in 2005.

Our contribution to this project comprised amongst other things results in the test of railway interlockings using TTCN-3 (Calamé et al., 2006a) as well as in the automated generation of TTCN-3 test cases from system specifications in Unified Modeling Language (UML), which has been described in Calamé et al. (2006b). This latter work is based on automatic test generation from formal specifications, which will be discussed in-depth in this thesis.

In the TT-Medal project, we did most of the research concerning our test-related research questions. A part of the work was also done in the scope of the Dutch *Basic Research in Informatics for Creating the Knowledge Society* (BRICKS) project (see BRICKS, online). This concerns mainly the research on model checking, but partially also that on testing.

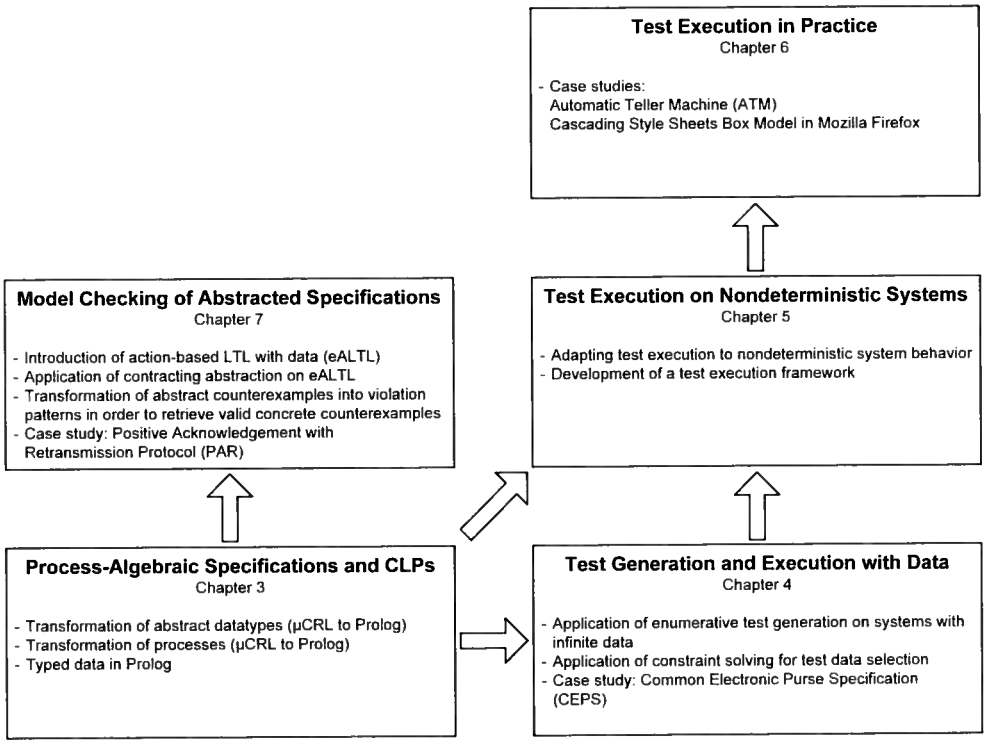

Figure 1.1: Structure of results

1.3.1 Contribution

In this thesis, we will consider research questions from the areas of software testing and model checking. Our research questions mainly focus on the problem, how

to handle systems with data from infinite domains in the several activities testing and model checking, and how to apply appropriate data selection while performing these activities. An overview of the results of this thesis is given in Figure 1.1. Overview on related work will be given in the respective chapters of this thesis. We can define three main questions, surveying the automatic generation and execution of test cases with data and the verification of abstracted system models:

1. How can test cases for a data-dependent system be generated using data abstraction techniques, so that data dependencies within this system are preserved for test data selection?
2. How can these test cases be generated and executed, so that they consider nondeterministic system behavior during test execution?
3. How can false negatives in model checking be utilized for the generation of real counterexamples?

Our solution to these questions is partially based on a uniform approach utilizing constraint solving techniques. For the specifications of the regarded systems, we use the process-algebraic specification language *micro Common Representation Language* (μCRL), as it has been defined by Groote and Ponse (1994). With μCRL, the behavior of a system can be defined using the common means of expression of process algebras (Fokkink, 2000). Furthermore, μCRL introduces data based on the concept of Abstract Datatypes (ADTs). Regarding specifications in μCRL and Constraint Logic Programs (CLPs), the fourth question arises regarding the correlation between these two:

4. How can process-algebraic specifications in μCRL be transformed to CLPs, such that their semantics is not impaired?

Question 1: Automatic Test Generation and Execution with Data

In the scope of our first two research questions, we work on the field of conformance testing. In a conformance test, an Implementation under Test (IUT) is tested for compliance with its specification. Such tests support measuring software quality mainly regarding the ISO 9126 category *functionality* (Hendriks et al., 2002). Test cases for conformance tests can be generated, for instance, using enumerative test case generators like Test Generation with Verification Techniques (TGV) developed by Jard and Jéron (2005), which is part of the Construction and Analysis of Distributed Processes (CADP) toolset. As enumerative model checkers, such test generators suffer from state space explosion. This problem can be circumvented by either using symbolic techniques, which use symbolic variables rather than enumerating all values of a particular datatype, or applying abstraction techniques.

In this thesis, we decided for the latter approach. We specify a system in μCRL and then abstract its data using a so-called *chaotic* data abstraction, in which all values of a datatype are reduced to a single constant for inputs and outputs of the system. From this abstracted system, we can then generate the control flow of the test using

TGV. The relevant data dependencies of inputs and outputs are preserved in a CLP. This CLP then serves as an oracle during test execution in order to receive test data for a test run.

Question 2: Test Execution on Nondeterministic Systems

After a test case has been generated, it is executed in parallel to the IUT. Depending on the reactions of the IUT on input from the tester, a verdict is assigned, which in short claims that the IUT is faulty or not. Based on this verdict, the development of the software iterates through a debugging cycle or goes on to the implementation of further parts or the release of the product.

However, if a system – either in its specification or in its implementation – is nondeterministic, verdicts are not always reliable. There are two possible reasons for the nondeterminism of a system: The first one is a specification that leaves certain decisions open to the implementation stage of the system, rather than choosing a particular decision already in the design stage. The implementation of such a system can still be a deterministic one. The second alternative is a system consisting of several components, which are executed in parallel and can send messages to the system's environment in any interleaving order.

Test generation from a system specification, however, selects a certain sequence of inputs to and outputs from the IUT before the execution together with appropriate testing data. For instance, the test of an Automatic Teller Machine (ATM) could expect a single 20 € banknote from the machine when withdrawing 20 € from a bank account. If the machine instead returns two 10 € banknotes, as an example for an implementation choice as mentioned above, the machine could then be wrongly declared as faulty, even if its specification *also* allows this alternative.

In this thesis, we introduce dynamic adaption of test execution to avoid this kind of wrong test verdicts. This approach can be compared to adaptive test cases (cf. Hierons, 2006) from the area of testing with Finite State Machines. In our approach, the sequence of actions and the appropriate test data are selected before test execution, too. This corresponds to the expectation of a single 20 € banknote in our example. However, instead of sticking to this selection, a digression of the IUT during test execution is detected and, if this is allowed by the specification, the sequence of actions and test data are adapted to the current situation and the test is executed further. Therefor, a new matching test trace is computed on the fly and appropriate test data is selected. In our example, it corresponds to an adaption, that the ATM returns two 10 € banknotes. By adaptation, we prevent a wrong test verdict. The test run from the example would thus not fail due to the emission of the two 10 € banknotes.

Question 3: Model Checking of Abstracted Specifications

In the scope of our third research question, we work in the field of model checking. As it had been described earlier in this chapter, this is a formal method to verify the correctness of a system with respect to a set of properties.

We consider enumerative model checking here, which suffers from state space explosion. For this reason, models and requirements must in many cases be abstracted prior to verification. This abstraction, however, makes model checking in some sense unprecise: A counterexample on the abstract level might not be reproducible on the concrete level of the system and properties. These counterexamples are known as *false negatives* (Clarke et al., 1999).

False negatives can in most cases be sorted out quite reliably, since it is in principle possible to verify their non-reproducibility. However, a false negative might still provide useful information on bugs. We provide a framework in this thesis to not sort out these false negatives completely, but to abstract further from them in order to find bugs in their peripherals. The approach is to take a false negative counterexample and to transform it into a more general *violation pattern*, which preserves the original property violation, but allows a wider set of possible traces. From this wider set, we might still be able to select another, reproducible counterexample using constraint solving techniques, and so reuse the otherwise unused knowledge from the false negative.

Question 4: Process-Algebraic Specifications and CLPs

The fourth research question arises from addressing the three other ones. Data selection for test execution, as well as the search for real counterexamples from false negatives in model checking, have both been approached using constraint solving techniques. The starting point for our research is the formal specification of a system. This specification is formulated in the process-algebraic specification μCRL (cf. Groote and Ponse, 1994).

We have thus to provide a theoretical underpinning for the transformation of μCRL specifications to CLPs. We do this in three parts: First of all, we provide a meta language on top of the constraint solver for convenience. Second, we provide a theory about mapping ADTs from their μCRL notation to that of the constraint solver, and finally, we provide a similar mapping for the behavioral part of μCRL specifications. While doing so, we have to preserve the semantics of μCRL in the CLP. This preservation will be proven for our theory.

Tooling

In the scope of this thesis, a number of tools has been realized, which allow the practical evaluation of the attained theoretical results. In particular, we developed tools to apply chaotic data abstraction to a given μCRL specification (cf. Chapter 4) and to transform μCRL specifications to CLPs in ECLiPSe Prolog (applied in all chapters, esp. see Chapter 3). Furthermore, to execute tests against a system with a nondeterministic specification or implementation, we developed the tool, Behavior-Adaptation in Testing (BAiT). It precomputes a trace through the system specification and tries to execute it. If the system diverts from this trace, BAiT tries to adapt its

execution accordingly. The tool will be introduced in Chapter 5 and will be applied to two case studies, an ATM and Mozilla Firefox in Chapter 6.

1.3.2 Detailed Structure of the Thesis

Here, we will discuss the structure of this work in detail. We will also name the exact contribution of the single chapters.

Chapter 2: In this chapter, we will define the preliminaries of formal specifications, as they are relevant for the remainder of this thesis. We will therefore introduce automata in their several flavors, and the process-algebraic specification language μCRL, also providing the necessary underpinnings in general algebra. Furthermore, we will introduce the UML as the language, which we use for illustrative purposes throughout the thesis. Finally, we will give a general introduction to the abstraction of systems.

Chapter 3: In this chapter, we will introduce constraint solving. We will use constraint solving throughout this thesis to determine data and traces, which are suitable for our purposes in testing and model checking a system.
The contribution of this chapter is a novel theory for the transformation of μCRL specifications into CLPs on the level of datatypes and behavior.

Chapter 4: In this chapter, we will introduce data abstraction as a means to facilitate the automatic generation and execution of conformance tests for systems with infinite data domains. Therefore, we will first introduce the theoretical fundaments of conformance tests, before discussing chaotic data abstraction and its application to test generation.
The contribution of this chapter is a theory, which allows to use data abstraction in the context of test generation based on Labeled Transition Systems (LTSs). In the course of test generation, our approach also preserves data interdependencies in a system.
The basis of this chapter is mainly formed by Calamé et al. (2005).

Chapter 5: In this chapter, we develop a practical realization of the above-named theoretical approach on test generation and execution.
The contribution of this chapter is a theory on test execution on nondeterministic IUTs with a discussion of the influence of quiescence on test execution. Furthermore, we present the tool BAiT, which has been developed as a practical realization of this theory.
The basis of this chapter is mainly formed by Calamé et al. (2007a).

Chapter 6: In this chapter, we present two different case studies for our test generation and execution approach. The first one is an academic behavior-oriented case study of a system with a nondeterministic specification. The second case study regards a real-life case testing an HTML rendering engine.
The contribution of this chapter is the practical show-case, how to apply the above-mentioned theory to practical software testing.

The basis of this chapter is formed by Calamé (2007) and Calamé and van de Pol (2008).

Chapter 7: In this chapter, we change our focus to model checking of abstracted systems. We first introduce the necessary preliminaries for Linear Temporal Logic (LTL) and its action-based counterpart. Then, we introduce data in what we name Extended Action-based Linear Temporal Logic (eALTL), and provide data abstraction for this logic. We then discuss the classes of resulting counterexamples, and provide a theory to widen the use of *false negative* counterexamples.
The contribution of this chapter is the introduction of data for action-based LTL, its abstraction and the development of an approach to also use *false negative* counterexamples in order to find faults in a system model.
The basis of this chapter is formed by an extended version of Calamé et al. (2007b).

Chapter 8: This chapter concludes the thesis and provides ideas for future work.

In order to understand this thesis, it is mandatory to read Chapters 2 and 3 in sequential order. The reader can then choose to read on with Chapters 4; 5 and 6, or to read Chapter 7 instead. Chapter 8 concludes both traces through this thesis. Each of the chapter also contains a discussion of related work.

Chapter 2

Specifications and Automata

Dem Anwenden muss das
Erkennen vorausgehen.

(Max Planck)

The fundamentals necessary to follow the rest of this thesis will be provided in this chapter. In this thesis, we test and model check systems, which are formally specified. The specifications that we work on are actually given in the micro Common Representation Language (μCRL), a process algebra with data (Groote and Ponse, 1994). Such specifications are in the course of our work transformed into a representation of their semantics, Input/Output Labeled Transition Systems (IOLTSs). In order to do so, we will in most cases have to apply abstractions.

In the remainder of this work, we will use Input/Output Symbolic Transition Systems (IOSTSs) illustrated in the notation of the UML instead of μCRL wherever possible, since they are easier to communicate. For this reason, we will in Section 2.1 introduce IOSTSs as a means of specification, and will define how IOLTSs resemble their semantics. In Section 2.2, we will introduce UML as a notation to specify the behavior and the structure of systems. In Section 2.3, we will introduce terms and algebras as the basis for the introduction of μCRL in Section 2.4. We finally give an introduction to abstractions in Section 2.5.

What the reader won't find here...

This chapter is organized as a general introduction into those notions, which are necessary to understand the *whole* thesis. However, there are also notions, which are necessary to understand a part of the thesis, because they are related exclusively to testing or model checking. Furthermore, there is a whole chapter exclusively dedicated to constraint solving.

In this chapter, we will thus spare out an introduction to constraint solving, which the reader will find in Chapter 3. Furthermore, we will postpone the introduction to testing theory to Chapter 4, and that to model checking-related logics to Chapter 7.

2.1 Specifications and Automata

2.1.1 Transition Systems in General

One possibility to specify a system is to define its behavior using a *transition system*. It allows us to describe a system's behavior graphically by using nodes for system states and edges for actions, which are either fired proactively by the system, or which are given as input to the system by its environment.

There are several classes of transition systems. On the one hand, we classify whether a transition system explicitly distinguishes system input and output actions (transition systems vs. input/output transition systems). On the other hand, we distinguish transition systems which handle data symbolically, i.e. as explicite variables, from those, which do not explicitly handle data at all (Symbolic Transition Systems (STSs) vs. LTSs). In this thesis, we will mainly handle input/output transition systems, in both their symbolic and labeled shape.

2.1.2 Symbolic Transition Systems

First, we will introduce the notion of IOSTSs. These are finite automata, which can describe the behavior of a system and the interrelation of its data. For reasons of simplicity, we want to consider only single-domain typing in this section.

Definition 2.1 (Valuation). Let $\mathbb{D}$ be a data domain, let Vn be a set of names and let $Var = \{x_1, \ldots, x_n\} \subseteq Vn$ be a set of variables. Let furthermore be $x_i : \mathbb{D}$ of domain $\mathbb{D}$ for $1 \leqslant i \leqslant n$ and let $v_i \in \mathbb{D}$ be a value in $\mathbb{D}$ for the same values of i. Then $\eta \subseteq Var \times \mathbb{D}$ with $\eta = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ is a *valuation* for Var. ■

An assignment of value $v \in \mathbb{D}$ to a variable $x \in Var$ with $x : \mathbb{D}$ within a valuation η will be denoted $\eta_{[x \mapsto v]}$.

Definition 2.2. Let S be a set. Then $S^* = \{(s_1, \ldots, s_n) | n \in \mathbb{N} \cup \{0\} \wedge s_1, \ldots, s_n \in S\}$. ■

Definition 2.3 (Expression). An *expression* is a general term which is constructed from names and operators. Valuations of variables are lifted to valuations of expressions in the standard way. ■

Definition 2.4 (Guard). Let $Exprs$ be a set of expressions. Then a guard is a boolean expression $g \in Exprs$. We will write $[\![g]\!]$ for the evaluation of g. ■

Definition 2.5 (IOSTS; cf. Jéron, 2004). Let Vn be a set of names, let G be a set of guards and let finally Val be a set of valuations. An IOSTS is a quintuple $\mathfrak{S} = (L, Var, A, E, (\ell_{init}, \eta_{init}))$ with

- L being a set of control states (*locations*),
- $Var \subseteq Vn$ being a set of variables in the system,
- A being a set of actions as defined later, and
- $E \subseteq L \times A \times L$ being a transition relation.

States in an IOSTS are elements of the set $L \times Val$ with the initial state $(\ell_{init}, \eta_{init})$ being the initial valuation η_{init} of all variables in Var at its initial location ℓ_{init}. ■

Definition 2.6. Actions and Edges in IOSTSs Let $\mathfrak{S} = (L, Var, A, E, (\ell_{init}, \eta_{init}))$ be an IOSTS, let G be a set of guards with $g \in G$, let $Exprs$ be a set of expressions and let $Events = Events_{in} \dot\cup Events_{out} \dot\cup \{\tau\}$ be a set of action names.

The set of actions $A = A_{in} \dot\cup A_{out} \dot\cup A_\tau$ is divided into input, output and internal actions as follows:

- $A_{in} \subseteq G \times Events_{in} \times Var^*$ is the set of input actions,
- $A_{out} \subseteq G \times Events_{out} \times Exprs^*$ is the set of output actions, and
- $A_\tau \subseteq G \times \{\tau\} \times (Var \times Exprs)^*$ is the set of internal actions.

An edge in an IOSTS is defined as a tuple $(\ell, \iota, \hat\ell) \in E$ with $\iota \in A$. In the remainder, we will write an edge $(\ell, \iota, \hat\ell)$ as $\ell \xrightarrow{\iota} \hat\ell$. The set of parameters of an action will be reduced to a single variable for convenience.

The set of actions A in $\mathfrak{S}$ is divided into three disjunct sets. This distinction has consequences for the labelling of edges in an IOSTS:

Input actions: Edges with input actions have the form $\ell \xrightarrow{g \triangleright ?s(x)} \hat\ell$. Receiving an event s with an actual parameter $v \in \mathbb{D}$ for a data domain $\mathbb{D}$ and $s : \mathbb{D}$ results in an update of the system's current valuation $\eta_{[x \mapsto v]}$ and a change of the actual location to $\hat\ell$.

Output actions: Edges with output actions have the form $\ell \xrightarrow{g \triangleright !s(e)} \hat\ell$. This means, that the event s is sent with an expression e as the actual value for $x : \mathbb{D}$ for a data domain $\mathbb{D}$ and $s : \mathbb{D}$. The location changes to $\hat\ell$.

Internal actions: which contains system-internal variable assignments. The appropriate edges have the form $\ell \xrightarrow{g \triangleright x := e} \hat\ell$. In case the transition fires, the system's current valuation is updated to $\eta_{[x \mapsto e]}$ and the actual location is changed to $\hat\ell$. The action name τ is left out when labelling an edge with an internal action.

■

When we regard actions in general without regarding any details, we will denote an action ι. Considering a system as a blackbox, input and output actions are *visible* while internal actions are *invisible*. The guard on an edge steers whether a particular action can be taken in a particular state of the system. If in a transition $\ell \xrightarrow{g \triangleright s(x)} \hat\ell$, g evaluates to $\top$, the transition can *fire*, i.e. the appropriate action can be taken. If the guard evaluates to $\bot$, it cannot. We will use the above-introduced syntax for edges, $\ell \xrightarrow{g \triangleright s(x)} \hat\ell$, in continuous text only. In graphical representations of system behavior, we will use UML state charts, as defined by the OMG (2005). Here, the syntax for edges is $\ell \xrightarrow{[g]s(x)} \hat\ell$. We will discuss the UML in more detail in Section 2.2.

Jéron (2004) treats the states of an IOSTS as a combination of variables, constants and communication parameters. We, by contrast, distinguish between a location and valuations in general. This does not limit the definition of IOSTSs: Variables, constants

and communication parameters contain valuations, and locations can be defined by a set of variables which by circular reasoning also contain values only. The initial condition of Jéron is reflected in the initial state of our definition of IOSTSs. In this point, our model of IOSTSs is limited w.r.t. to the one of Jéron, since the latter allows more than just one initial state of a system. Our further work, however, will not be negatively affected by this restriction, since the systems that we work on are specified in a process algebra, which also allows only one initial state for a system.

The introduction of distinct locations, however, allows an easier handling of IOSTSs. These can now be described as directed graphs, where L is the set of nodes and E is the set of edges between nodes.

2.1.3 Labeled Transition Systems

Now, we define IOLTSs. Unlike IOSTSs, data cannot be handled symbolically in IOLTSs. This means, that, while variables in an IOSTS can reach different values in one location of this IOSTS, this is not possible in IOLTSs. Rather, for each combination of single data values, a single state has to be introduced. Furthermore, a distinction between actions and action parameters in transitions of the system does not take place.

Definition 2.7 ((Deterministic) IOLTS). An IOLTS is a quadruple $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{init})$ where

- $\Sigma \neq \emptyset$ is a set of states,
- $\Lambda = \Lambda_{in} \dot\cup \Lambda_{out} \dot\cup \{\tau\}$ is a set of action labels,
- $\Delta \subseteq \Sigma \times \Lambda \times \Sigma$ is a transition relation, and
- $\sigma_{init} \in \Sigma$ is the initial state.

The set of labels Λ consists of the three disjunct subsets, Λ_{in}, Λ_{out}, and $\{\tau\}$ denoting input, output and internal actions.

An IOLTS is *deterministic* if and only if there is at most one outgoing transition for each action label $\lambda \in \Lambda$ in each state $\sigma \in \Sigma$. ■

Definition 2.8 (Input-complete IOLTS). An IOLTS $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{init})$ with $\Lambda = \Lambda_{in} \dot\cup \Lambda_{out} \dot\cup \{\tau\}$ is *input-complete*, if $\forall \sigma \in \Sigma . \forall \lambda \in \Lambda_{in} . \exists \hat\sigma \in \Sigma : \sigma \xrightarrow{\lambda} \hat\sigma \in \Delta$. ■

An IOLTS can also be described as a directed graph, where Σ is the set of nodes and Δ the set of edges between nodes. Each transition is labeled with an action label; for a transition $(\sigma, \lambda, \hat\sigma) \in \Delta$, we write more suggestively $\sigma \xrightarrow{\lambda} \hat\sigma$. An IOLTS defines the semantics of an IOSTS as is given by the semantic transformation rules in Table 2.1.

Definition 2.9 (Semantics of an IOSTS). Let $Events$ be a set of action names, let G be a set of guards and let Val be a set of valuations. Then, the semantics of an IOSTS $\mathfrak{S} = (L, Var, A, E, (\ell_{init}, \eta_{init}))$ is defined by the IOLTS $[\![\mathfrak{S}]\!] = \mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{init})$ with

- $\Sigma \subseteq L \times Val$ the set of reachable states in $\mathfrak{S}$,
- $\Lambda \subseteq Events \times Val$ the set of actions with evalulated action parameters as the set of labels,

- Δ the set of edges from $\mathfrak{G}$ transformed by the rules from Table 2.1 with $g \in G, s \in Events, \ell, \hat{\ell} \in L$ and $v \in \mathbb{D}_1$ for $s(v)$ with $s : \mathbb{D}_1$, and
- $\sigma_{init} = (\ell_{init}, \eta_{init})$ the initial state.

■

$$\text{II-a}\ \frac{\ell \xrightarrow{g \triangleright ?s(x)} \hat{\ell} \in E \quad [\![g]\!]_{\eta_{[x \mapsto v]}} = \top}{(\ell, \eta) \xrightarrow{?s(v)} (\hat{\ell}, \eta_{[x \mapsto v]}) \in \Delta}$$

$$\text{II-b}\ \frac{\ell \xrightarrow{g \triangleright !s(e)} \hat{\ell} \in E \quad [\![g]\!]_{\eta} = \top \quad [\![e]\!]_{\eta} = v}{(\ell, \eta) \xrightarrow{!s(v)} (\hat{\ell}, \eta) \in \Delta}$$

$$\text{II-c}\ \frac{\ell \xrightarrow{g \triangleright x := e} \hat{\ell} \in E \quad [\![g]\!]_{\eta} = \top \quad [\![e]\!]_{\eta} = v}{(\ell, \eta) \xrightarrow{\tau} (\hat{\ell}, \eta_{[x \mapsto v]})}$$

Table 2.1: Semantic transformation of IOSTSs to IOLTSs

The behavior of an IOLTS is given by sequences of states and transitions $\pi = \sigma_{init} \xrightarrow{\lambda_1} \sigma_1 \xrightarrow{\lambda_2} \ldots$ starting from the initial state. In traces, the states are projected out, i.e. $[\![\mathfrak{M}]\!]_{traces} \subseteq \Lambda^*$.

Definition 2.10 (Trace in IOLTS). A *trace* π of $\mathfrak{M}$ is a mapping $\pi_\lambda : N \to \Lambda$, where either $N = \{1, 2, \ldots, n\}$ is finite or $N = \mathbb{N} \setminus \{0\}$, and there exists a mapping $\pi_\sigma : N \to \Sigma$ s.t. $(\pi_\sigma[i-1] \xrightarrow{\pi_\lambda[i]} \pi_\sigma[i]) \in \Delta$ for all $i \in N$. If $N = \mathbb{N}$, trace π is called an *infinite* trace; otherwise, it is called *finite*. The *length* of π for finite traces is defined as $|N|$ and referred to as $|\pi|$. ■

In the further text, we refer to the set of traces in $\mathfrak{M}$ as $[\![\mathfrak{M}]\!]_{traces}$. For reasons of readability, we will furthermore write π_λ only, if we have to explicitly distinguish it from π_σ. Otherwise, we will simply write π for π_λ. The relation $after$ provides possible options for actions after a given trace in the system.

Definition 2.11 (Relation $after$). Let $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{init})$ be an LTS. The relation $after$ is defined for states and action labels as $\Sigma \times \Lambda \to 2^\Sigma$ with σ $after$ λ being the set of states which can be reached from σ by a transition labeled with λ: σ $after$ $\lambda = \{\hat{\sigma} \in \Sigma | \sigma \xrightarrow{\lambda} \hat{\sigma} \in \Delta\}$.

For traces and action labels, the relation $after$ is defined as $[\![\mathfrak{M}]\!]_{traces} \times \Lambda \to 2^\Sigma$ with π $after$ λ being the set of states which can be reached by a transition labeled with λ after trace π: π $after$ $\lambda = \{\hat{\sigma} \in \Sigma | \sigma_{init} \xrightarrow{\pi} \sigma \xrightarrow{\lambda} \hat{\sigma} \in \Delta\}$. ■

2.2 The Unified Modeling Language

The Unified Modeling Language (UML) has been developed and standardized by the Object Management Group (OMG) as an industrial approach to the specification of software systems. The language itself is defined by a meta model (OMG, 2005) and can be extended by so-called *profiles*. A relevant example of such a profile is the *UML 2 Testing Profile* (OMG, 2003; Baker et al., 2008).

In this thesis, we will use a few of the concepts of the UML for the graphical representation of system behavior and the structure of systems. In particular, we will use an adapted notion of UML State Machine Diagrams to design STSs, UML Sequence Diagrams to design the interaction of different components in a system as well as UML Component and Class Diagrams to design the internal structure of a system. We will introduce these diagram types in the remainder of this section.

2.2.1 System Behavior

System behavior is in this thesis designed using State Diagrams to define the complete behavior of a system and Transition Diagrams to give an example for the interaction of different parts of a system.

State Machine Diagrams This diagram type is rooted in the theory of finite automata which were introduced to the development of software systems by Harel (1987). Here, we give an example for an STS together with the simplified graphical UML-based notation, which we will use further in this thesis.

Example 2.12. Let $\mathfrak{S} = (L, Var, A, E, (\ell_{init}, \eta_{init}))$ be an IOSTS with

- $L = \{\ell_1, \ell_2, \ell_3\}$ the set of locations,
- $Var = \{x\}$ the set of variables,
- $A = A_{in} \cup A_{out}$ with $A_{in} = \{a\}$ and $A_{out} = \{b, c\}$ the set of actions,
- $E = \{\ell_1 \xrightarrow{?a(x)} \ell_2, \ell_2 \xrightarrow{x>5 \triangleright !b(x)} \ell_3, \ell_2 \xrightarrow{x \leqslant 5 \triangleright !c(x)} \ell_3\}$ the set of edges, and
- $(\ell_1, \{x \mapsto \bot\})$ the initial state of the system.

The graphical notation used in this thesis for the above-defined system will be the one of Figure 2.1. Locations ℓ_1 and ℓ_2 are represented by the round rectangles with labels 1 and 2. The third location, ℓ_3, is represented by the circled dot on the right side of the figure, which is a *final state*[1]. Notice, that ℓ_1 is preceded by a dot in the figure, which is an *initial state*. Initial states in UML state machine diagrams are *pseudo states*. We will in the remainder of this thesis use the transition from the initial state of the diagram to the actual initial location of the system for the initialization of internal variables from Var.

[1] The word "state" is the UML nomenclatura. In this context, it means "locations", rather than "states" in an LTS.

As a difference to the official UML standard, transitions are labeled as defined earlier in this chapter. The only difference is the use of square brackets instead of a single triangle for the guard of a transition. Branches in a system behavior, induced by guards, are preceded by a pseudo state for a *choice*, depicted by an empty diamond. Its ingoing transition is unlabeled.

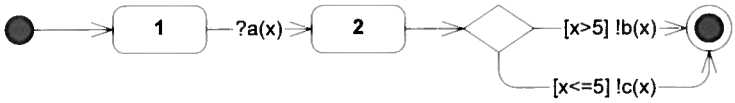

Figure 2.1: A UML State Machine Diagram

Sequence Diagrams UML Sequence Diagrams can be compared with Message Sequence Charts (MSCs), as they have been defined by the ITU-T (2005). They are used to give an example for the interaction of several components of one or more systems. A sequence diagram has two dimensions, one dimension distinguishing the participating components (horizontally), while the second dimension is the elapsing time during interaction (vertically).

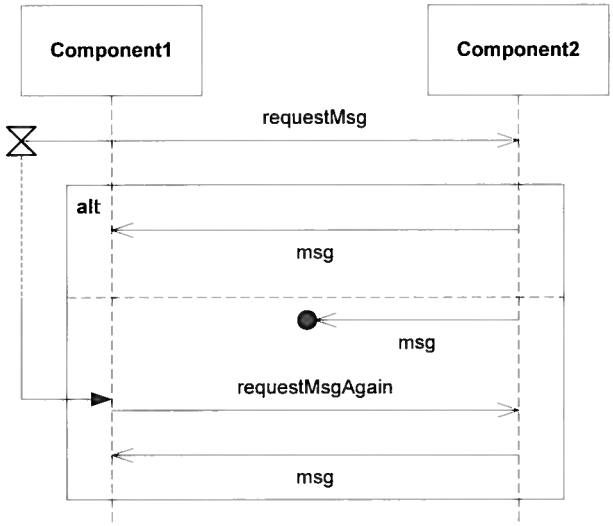

Figure 2.2: A UML Sequence Diagram

Example 2.13. Let us take Figure 2.2 as an example. The diagram shows two components, `Component1` and `Component2`, in interaction. First, `Component1` sends a message to `Component2`. Afterwards, an alternative block is entered. In the first alternative, the message sent from `Component2` is received by `Component1`, while in the second alter-

native, it is lost and has to be resent. Losing a message is depicted by a filled dot in the diagram, which receives this message.

Alternatives in UML Transition Diagrams are represented by a rectangle, which is labeled with **alt** and which covers a subset of the communication acts in the diagram. The two alternatives are divided by a horizontal dashed line within the alternative construct.

Finally, we have extended the notation of UML Transition Diagrams by the concept of timeouts as you can also find it in the specification of the UML 2 Testing Profile by the OMG (2003). In our example, a timer is started when message `requestMsg` is sent. Starting a timer is represented by a stylized hourglass. If `Component1` receives the message `msg` within the duration of the timer, the timer is cancelled. `Component1` sends the message `requestMsgAgain`, when the timer has timed out, depicted with an arrow pointing from the timer at the life line of the component holding this timer. The concept of timeout, however, had to be added to the standard notation of the UML, since it is originally not supported, even though timing constraints can very well be modeled in UML Transition Diagrams. IOSTSs and IOLTS also do not support timers perse, however, there exist several extensions which we will discuss in Section 5.1.2.

2.2.2 System Structure

A system does not only exhibit behavior, but has also entities which implement this behavior. While STSs and LTSs concentrate on the behavior only, there are specification languages which also allow to consider the structure of entities behind this behavior, like, for instance, process algebras with a concept of processes as proactive system entities or object-oriented specification languages. One such language is the UML, which provides Component and Class Diagrams to design the structural aspects of a system and even of the UML itself. Here, we give a short introduction to these diagram types.

Example 2.14. The example in Figure 2.3 specifies a component `MyComponent`. A component is represented as a *classifier* (visually a rectangle) with the stereotype «`component`» and an optional graphical stereotype as depicted in the figure. Stereotypes denote the extension of existing modelling elements in UML mainly providing a usage context of the respective element. In our example, the component is realized by a class `MyClass`, which is also represented by a classifier. A component is a coarse-grain design element, which abstracts from its realizing classifiers. A class is a fine-grain design element, which combines a set of operations (actions) and attributes. A component can be realized by other classifiers, i.e., for instance, classes and other components.

Component and Class Diagrams divide operations into inputs, outputs and internal operations. While internal operations are defined in classes, input and output operations can be extracted to interfaces. Those are classifiers with the stereotype «`interface`». We define two interfaces in our example, of which `ProvidedInterface` is implemented or provided, resp., by `MyClass` and thus by `MyComponent`, while `Re-`

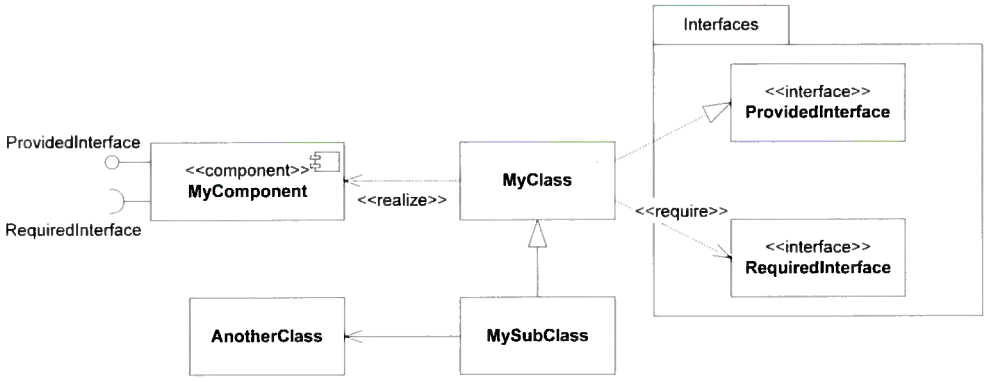

Figure 2.3: A UML Component/Class Diagram

`quiredInterface` is required from other components connecting with `MyComponent` or `MyClass`, resp.

There are two ways to define the relation between a classifier and its interfaces. The extended way is depicted on the right side of Figure 2.3. Here we explicitly define the interfaces and associate the providing or requiring class with an explicit implementation of the requirement dependency (the arrows from class `MyClass` to the interfaces). A shorthand notation is depicted on the left side of the figure. It is the so-called *lollipop notation*, where a provided interface is depicted as an empty circle at a handle. A required interface, a so-called *socket*, is in this notation represented by a open semicircle at a handle.

Figure 2.3 contains a few more concepts of UML Component and Class Diagrams. The first one is packages, which provide a means to group packageable elements like classifiers. In our example, the two interfaces `ProvidedInterface` and `RequiredInterface` are grouped in the package `Interfaces`. The second concept is that of inheritance. In this example, the class `MySubClass` inherits from `MyClass` and by that extends the operations and attributes of its parent class. Finally, class `MySubClass` has a navigable association with a class `AnotherClass`, which forms, for instance, an attribute of `MySubClass`.

While dependencies (realize and require) and implementations are depicted using dashed arrows, inheritance and associations are depicted with straight arrows[2]. The navigation direction in our example points out, that `MySubClass` "knows" an instance of `AnotherClass`, while instances of `AnotherClass` cannot find out by whom they are held. By adding an empty or filled diamond on the side of that class, which is attributed with the other class, this association can be made even stronger. We will not discuss these details here.

[2]For technical reasons, this difference might not always be visible in this thesis.

2.3 Terms and Algebras

In this section, we will introduce a number of constructs for multi-sorted algebras. Based on these algebras we can then introduce Abstract Datatypes (ADTs) in the process algebraic specification language μCRL. These constructs will first be introduced by means of their syntax, and afterwards be lifted to their semantics.

2.3.1 The Syntax: Signatures and Terms

A signature forms the foundation of terms in an algebra.

Definition 2.15 (Signature; cf. Ihringer, 1993). A multisorted signature is a triple $\mathfrak{Z} = (\mathcal{S}, \mathcal{F}, \sigma)$ with

- $\mathcal{S}$ a set of sorts,
- $\mathcal{F}$ a set of operation symbols and
- $\sigma : \mathcal{F} \to \mathcal{S}^* \times \mathcal{S}$ a function, which assigns to an operation $f \in \mathcal{F}$ the sorts of its arguments and its own sort.

■

Terms form the data and function notation of an algebra.

Definition 2.16 (Term; ibid, extended). Let $\mathfrak{Z} = (\mathcal{S}, \mathcal{F}, \sigma)$ be a multisorted signature. A set of (open) terms is defined as $\mathfrak{T}(F, X)$ with F a set of operations over $\mathcal{F}$ and $X = (X_{\mathbb{S}})_{\mathbb{S} \in \mathcal{S}}$ a family of variable sets over the sorts from $\mathcal{S}$. For the elements from F, the assignment of sorts of arguments and the operations themselves is defined by σ.

Let $\mathbb{S}_1, \ldots, \mathbb{S}_n \in \mathcal{S}$. A term $t \in \mathfrak{T}(F, X)$ is an element of the smallest set, for which the following holds:

1. $X_{\mathbb{S}} \subseteq \mathfrak{T}(\mathcal{F}, X)$.
2. $\forall f \in \mathcal{F}. \forall t_i \in \mathfrak{T}(\mathcal{F}, X) : \sigma(f) = \mathbb{S}_1 \times \ldots \times \mathbb{S}_n \to \mathbb{S} \Longrightarrow f(t_1, \ldots, t_n) \in \mathfrak{T}(\mathcal{F}, X)$.

■

A term is named *closed*, if it does not contain any variables, otherwise it is named *open*. In the further text, we will refer to variables with x, to operation symbols with f, g, and to terms with t, u, using an index if necessary.

2.3.2 The Semantics: Algebras for μCRL

The next definition addresses the assignment of elements from an algebra to variables and multisorted algebras. Those address the semantics behind the earlier-defined terms.

Definition 2.17 ($\mathcal{F}$-Algebra; cf. Bouma, 1991). Let $\mathfrak{Z} = (\mathcal{S}, \mathcal{F}, \sigma)$ be a multisorted signature. Let furthermore be X be a set of variables. Then $\mathfrak{A} = (A, I_{\mathfrak{A}})$ is an algebra over this signature if

- A the algebra's *carrier set* with $A = (A_{\mathbb{S}})_{\mathbb{S}\in\mathcal{S}}$, and
- $\vartheta : X \to A$ is an *assignment* of elements from A to variables from X, then
- $I_{\mathfrak{A}}$ is an *interpretation function* for elements of $\mathcal{F}$ as follows:

$$f \in \mathcal{F} \text{ with } f : \mathbb{S}_1 \times \ldots \times \mathbb{S}_n \to \mathbb{S} \Longrightarrow I_{\mathfrak{A}}(f) : A_{\mathbb{S}_1} \times \ldots \times A_{\mathbb{S}_n} \to A_{\mathbb{S}}$$

The interpretation of a term $f(t_1, \ldots, t_n)$ with the above-mentioned signature is defined inductively as:

$$\begin{aligned} I^{\vartheta}_{\mathfrak{A}}(f(t_1, \ldots, t_n)) &= I^{\vartheta}_{\mathfrak{A}}(f)(I_{\mathfrak{A}}(t_1), \ldots, I^{\vartheta}_{\mathfrak{A}}(t_n)) \\ I^{\vartheta}_{\mathfrak{A}}(x) &= \vartheta(x) \end{aligned}$$

■

In the remainder of this thesis, we will write $\mathbb{D}_i$ for a set $A_{\mathbb{S}_i}$. For an interpretation $I^{\vartheta}_{\mathfrak{A}}(f) : A_{\mathbb{S}_1} \times \ldots \times A_{\mathbb{S}_n} \to A_{\mathbb{S}}$, we also write $[\![f]\!] : \mathbb{D}_1 \times \ldots \times \mathbb{D}_n \to \mathbb{D}$.

The attempt to find solutions for interdependent variables of infinite data domains is mainly used for the domains of the natural numbers $\mathbb{N}$ and for boolean conditions of the domain $\mathbb{B}$=$\{\top,\bot\}$. For this reason, we will at this point exemplarily define a boolean algebra for μCRL, $\mathfrak{B}_\mu$, as well as an algebra for natural numbers, $\mathfrak{N}_\mu$.

Example 2.18. The boolean algebra, which we define here, will consist of constants for the boolean values $\top$ and $\bot$, and will furthermore contain the standard boolean operations $\neg$, $\wedge$, $\vee$ and equivalence in their default semantics.

Definition 2.19 (Signature for booleans in μCRL). The signature of booleans in μCRL is defined as

$$\mathfrak{Z}_{\mathrm{Bool}_\mu} = (\mathrm{Bool}_\mu, C_{\mathrm{Bool}_\mu} \cup M_{\mathrm{Bool}_\mu}, \sigma_{\mathrm{Bool}_\mu})$$

with $C_{\mathrm{Bool}_\mu} = \{T_\mu, F_\mu\}$ and $M_{\mathrm{Bool}_\mu} = \{eq_\mu, and_\mu, or_\mu, not_\mu\}$. The signature of functions $\sigma_{\mathrm{Bool}_\mu}$ is defined as

$$\begin{aligned} T_\mu &: \to \mathrm{Bool}_\mu \\ F_\mu &: \to \mathrm{Bool}_\mu \\ eq_\mu &: \mathrm{Bool}_\mu \times \mathrm{Bool}_\mu \to \mathrm{Bool}_\mu \\ and_\mu &: \mathrm{Bool}_\mu \times \mathrm{Bool}_\mu \to \mathrm{Bool}_\mu \\ or_\mu &: \mathrm{Bool}_\mu \times \mathrm{Bool}_\mu \to \mathrm{Bool}_\mu \\ not_\mu &: \mathrm{Bool}_\mu \to \mathrm{Bool}_\mu \end{aligned}$$

■

Definition 2.20 (A boolean algebra for μCRL). A boolean algebra for μCRL is defined as a tuple

$$\mathfrak{B}_\mu = (\mathbb{B}, \{[\![T]\!]_{\mathfrak{B}_\mu}, [\![F]\!]_{\mathfrak{B}_\mu}, [\![eq]\!]_{\mathfrak{B}_\mu}, [\![and]\!]_{\mathfrak{B}_\mu}, [\![or]\!]_{\mathfrak{B}_\mu}, [\![not]\!]_{\mathfrak{B}_\mu}\}).$$

Let $b_1, b_2 \in \mathbb{B}$ be variables. Then the interpretations $I^{\vartheta}_{\mathfrak{B}_\mu}$ are defined as follows:

$$\begin{aligned}
[\![T_\mu]\!]_{\mathfrak{B}_\mu} &\equiv \top \\
[\![F_\mu]\!]_{\mathfrak{B}_\mu} &\equiv \bot \\
[\![eq_\mu]\!]_{\mathfrak{B}_\mu}(b_1, b_2) &\equiv (b_1 \wedge b_2) \vee (\neg b_1 \wedge \neg b_2) \\
[\![and_\mu]\!]_{\mathfrak{B}_\mu}(b_1, b_2) &\equiv b_1 \wedge b_2 \\
[\![or_\mu]\!]_{\mathfrak{B}_\mu}(b_1, b_2) &\equiv b_1 \vee b_2 \\
[\![not_\mu]\!]_{\mathfrak{B}_\mu}(b_1) &\equiv \neg b_1
\end{aligned}$$

■

Example 2.21. For the algebra for natural numbers, we define a constant for the basic value 0 as well as a constructor S for all other values of $\mathbb{N}$. Furthermore, we define two of the basic computation operations, $+$ and $\cdot$, and the comparison operations $=$, $>$, $\geqslant$, $<$ and $\leqslant$ in their default semantics.

Definition 2.22 (Signature for natural numbers for μCRL). The signature of natural numbers for μCRL is defined as

$$\Im_{Nat_\mu} = (Nat_\mu \cup Bool_\mu, C_{Nat_\mu} \cup M_{Nat_\mu}, \sigma_{Nat_\mu})$$

with $C_{Nat_\mu} = \{0_\mu, S_\mu\}$ and $M_{Nat_\mu} = \{add_\mu, mult_\mu, eq_\mu, gt_\mu, ge_\mu, lt_\mu, le_\mu\}$. The signature of functions σ_{Nat_μ} is defined as

$$\begin{aligned}
0_\mu &: \rightarrow Nat_\mu \\
S_\mu &: Nat_\mu \rightarrow Nat_\mu \\
add_\mu &: Nat_\mu \times Nat_\mu \rightarrow Nat_\mu \\
mult_\mu &: Nat_\mu \times Nat_\mu \rightarrow Nat_\mu \\
eq_\mu &: Nat_\mu \times Nat_\mu \rightarrow Bool_\mu \\
gt_\mu &: Nat_\mu \times Nat_\mu \rightarrow Bool_\mu \\
ge_\mu &: Nat_\mu \times Nat_\mu \rightarrow Bool_\mu \\
lt_\mu &: Nat_\mu \times Nat_\mu \rightarrow Bool_\mu \\
le_\mu &: Nat_\mu \times Nat_\mu \rightarrow Bool_\mu
\end{aligned}$$

■

Definition 2.23 (Algebra for natural numbers (μCRL)). An algebra for natural numbers is defined as a tuple

$$\begin{aligned}
\mathfrak{N}_\mu = \quad (\mathbb{N} \cup \mathbb{B}, \quad & \{[\![0_\mu]\!]_{\mathfrak{N}_\mu}, [\![S_\mu]\!]_{\mathfrak{N}_\mu}, [\![add_\mu]\!]_{\mathfrak{N}_\mu}, [\![mult_\mu]\!]_{\mathfrak{N}_\mu}, [\![eq_\mu]\!]_{\mathfrak{N}_\mu}, [\![gt_\mu]\!]_{\mathfrak{N}_\mu}, \\
& [\![ge_\mu]\!]_{\mathfrak{N}_\mu}, [\![lt_\mu]\!]_{\mathfrak{N}_\mu}, [\![le_\mu]\!]_{\mathfrak{N}_\mu}\}).
\end{aligned}$$

Let $n_1, n_2 \in \mathbb{N}$ be variables. Then, the interpretations $I^{\vartheta}_{\mathfrak{N}_\mu}$ are defined as follows:

$$\begin{aligned}
[\![0_\mu]\!]_{\mathfrak{N}_\mu} &\equiv 0 \\
[\![S_\mu]\!]_{\mathfrak{N}_\mu}(n_1) &\equiv n_1 + 1 \\
[\![add_\mu]\!]_{\mathfrak{N}_\mu}(n_1, n_2) &\equiv n_1 + n_2 \\
[\![mult_\mu]\!]_{\mathfrak{N}_\mu}(n_1, n_2) &\equiv n_1 \cdot n_2 \\
[\![eq_\mu]\!]_{\mathfrak{N}_\mu}(n_1, n_2) &\equiv n_1 = n_2 \\
[\![gt_\mu]\!]_{\mathfrak{N}_\mu}(n_1, n_2) &\equiv n_1 > n_2 \\
[\![ge_\mu]\!]_{\mathfrak{N}_\mu}(n_1, n_2) &\equiv n_1 \geqslant n_2 \\
[\![lt_\mu]\!]_{\mathfrak{N}_\mu}(n_1, n_2) &\equiv n_1 < n_2 \\
[\![le_\mu]\!]_{\mathfrak{N}_\mu}(n_1, n_2) &\equiv n_1 \leqslant n_2
\end{aligned}$$

■

2.4 Specifications in μCRL

The language μCRL, defined by Groote and Ponse (1994), is a process-algebraic specification language with data. Compared to other specification languages like χ (Van Beek et al., 2006) or Focus (Broy, 1998), μCRL is a purely oriented on the process as such, defining neither objects nor channels for particular system components. The system is thus specified as a whole.

Definition 2.24 (μCRL specification; Groote and Ponse, 1994). A μCRL specification is defined as a pair $\mathfrak{S} = (\mathfrak{Z}, \mathcal{E})$ over its signature $\mathfrak{Z}$ and a set of equations $\mathcal{E}$. ■

As defined by Groote and Ponse (1994), a μCRL specification $\mathfrak{S}$ is given by a signature $\mathfrak{Z}(\mathfrak{S}) = (\mathcal{S}, \mathcal{F}, \mathcal{A}, \mathcal{C}, \mathcal{P})$. It specifies an open system that communicates with its environment. The set $\mathcal{S}$ defines a set of sorts, $\mathcal{F}$ defines a set of functions, composed from constructors and maps (μCRL-"functions"; cf. Definition 2.25). Furthermore, the signature contains a set of actions $\mathcal{A}$, communication definitions $\mathcal{C}$, defining which action communicates with which other action, and a set of processes $\mathcal{P}$. These processes then define the system's behavior. The entry point to the system behavior is a distinct process initialization ($\mathbf{init}$).

For μCRL specifications, there exists a normal form, the *linearized* specification, which we will discuss in Section 2.4.2. In this form, all processes are combined to a single one, whose control flow contains all possible interleavings of actions, which happen in parallel, limited by the synchronization of processes by communication.

2.4.1 Data in μCRL

μCRL is a process algebra with data. A μCRL specification defines a process algebra with a signature of which we first want to regard only the fragment that is relevant for the definition of ADTs. Data manipulation in μCRL is based on term rewriting (Bergstra et al., 1989).

Definition 2.25 (ADT-relevant part of μCRL specifications; Groote and Ponse, 1994). Let $\mathfrak{Z} = (\mathcal{S}, \mathcal{F}, \sigma)$ be a multisorted signature and let $\mathfrak{Z}_\mu = (\mathcal{S}_\mu, \mathcal{F}_\mu, \emptyset, \emptyset, \emptyset)$ be the ADT-relevant fragment of the corresponding *signature* of a μCRL specification as defined by Groote and Ponse (1994). Then $\mathcal{S}_\mu = \mathcal{S}$ and $\mathcal{F}_\mu \subseteq \mathcal{F} \times \sigma$ with $\mathcal{F}_\mu = C \dot{\cup} M$ being the union of a set of constructors C, i.e. terminal elements of the sorts from $\mathcal{S}_\mu$, and a set of functions (*maps*, M), for which term rewrite rules exist (see Definition 2.27). ■

Since we will only consider signatures of μCRL specifications in the remainder, we will skip the subscript μ from now on. The set $\mathcal{S}$ defines a set of datatypes for the declaration of variables. Each sort $\mathbb{S} \in \mathcal{S}$ consists of a set of constructors, which have the form $c :\rightarrow \mathbb{S}$ or $c : \mathbb{S}_1 \times \ldots \times \mathbb{S}_n \rightarrow \mathbb{S}$, resp., with $\mathbb{S}_1, \ldots, \mathbb{S}_n \in \mathcal{S}$. These constructors are used to form typed values v of sort $\mathbb{S}$.

In $\mathcal{F}$, functions of the form $f :\rightarrow \mathbb{S}$ or $f : \mathbb{S}_1 \times \ldots \times \mathbb{S}_n \rightarrow \mathbb{S}$, resp., are declared. Each of these functions is defined by one or more axioms on values of sorts $\mathbb{S}_1, \ldots, \mathbb{S}_n$. These axioms have the form $s = t$ where s and t are equally typed terms formed by any valid combination of typed variables and function symbols.

Definition 2.26 (Constructor terms, function terms and values). Let X be a set of variables and let $\mathfrak{Z}_\mu = (\mathcal{S}_\mu, \mathcal{F}_\mu, \emptyset, \emptyset, \emptyset)$ be the ADT-relevant fragment of the signature of a μCRL specification with $\mathcal{F}_\mu = C \dot{\cup} M$. Then we will name a term t

- a *constructor term* t if $t \in \mathfrak{T}(C, X)$,
- a *function term* $f(t_1, \ldots, t_n)$ if $f \in M \wedge t_i \in \mathfrak{T}(C, X)$ for $i \in \{1, \ldots, n\}$, and
- a *value* if $t \in \mathfrak{T}(C, \emptyset)$.

■

The language μCRL supports data which is represented by terms. Data manipulation in μCRL is built upon term rewriting. Term rewriting again is driven by equations. An equation is denoted $f(t_1, \ldots, t_m) = g(u_1, \ldots, u_n)$.

Definition 2.27 (Equation). Let $\mathfrak{Z} = (\mathcal{S}, \mathcal{F}, \emptyset, \emptyset, \emptyset)$ be the signature of a μCRL specification with $\mathcal{F} = C \dot{\cup} M$and let $\mathcal{E}$ be a set of equations. An equation $\varepsilon = (X, t, u) \in \mathcal{E}$ is a tuple with

- X the family of variables, over which the equation is defined,
- $t \in \mathfrak{T}_\mathbb{S}(\mathcal{F}, X)$ is the left hand side of the equation with $t = f(t_1, \ldots, t_m)$, $f \in M$ and t_i being function terms, and
- $u \in \mathfrak{T}_\mathbb{S}(\mathcal{F}, X)$ is its right hand side with $u = g(u_1, \ldots, u_n)$, g and u_i being constructor terms, function terms or values,

for some sort $\mathbb{S}$. ■

Figure 2.4 shows the definition of a sort Bool as it had been defined earlier in Example 2.18, but now in the syntax of μCRL. In doing so, b is defined as a variable of sort Bool (see **var**).

$$
\begin{array}{lll}
\textbf{sort} & \multicolumn{2}{l}{\mathrm{Bool}} \\
\textbf{func} & \multicolumn{2}{l}{T :\rightarrow \mathrm{Bool} \qquad F :\rightarrow \mathrm{Bool}} \\
\textbf{map} & \multicolumn{2}{l}{\mathrm{and} : \mathrm{Bool} \times \mathrm{Bool} \rightarrow \mathrm{Bool}} \\
 & \multicolumn{2}{l}{\mathrm{or} : \mathrm{Bool} \times \mathrm{Bool} \rightarrow \mathrm{Bool}} \\
 & \multicolumn{2}{l}{\mathrm{eq} : \mathrm{Bool} \times \mathrm{Bool} \rightarrow \mathrm{Bool}} \\
 & \multicolumn{2}{l}{\mathrm{not} : \mathrm{Bool} \rightarrow \mathrm{Bool}} \\
\textbf{var} & \multicolumn{2}{l}{b : \mathrm{Bool}} \\
\textbf{rew} & \mathrm{and}(T, b) = b & \mathrm{and}(b, T) = b \quad \mathrm{and}(F, b) = F \quad \mathrm{and}(b, F) = F \\
 & \mathrm{or}(T, b) = T & \mathrm{or}(b, T) = T \quad \mathrm{or}(F, b) = b \quad \mathrm{or}(b, F) = b \\
 & \mathrm{eq}(T, T) = T & \mathrm{eq}(F, T) = F \quad \mathrm{eq}(T, F) = F \quad \mathrm{eq}(F, F) = T \\
 & \mathrm{not}(T) = F & \mathrm{not}(F) = T
\end{array}
$$

Figure 2.4: Datatype for booleans

2.4.2 Behavior in μCRL

Basic means of expression for behavior specification The language μCRL defines several operators for the composition of actions, guards and processes to the complete behavior of a system. The most basic ones are $\iota_1 \cdot \iota_2$, which forms a sequence of the two actions ι_1 and ι_2, and $\iota_1 + \iota_2$, which defines a nondeterministic choice between ι_1 and ι_2. A deterministic choice between two actions based on a guard is defined by $\iota_1 \lhd g \rhd \iota_2$, which reads as *if* g *then* ι_1 *else* ι_2. The summation operator $\sum$ allows to introduce local variables, but is also used as an abbreviation for a number of $+$-operations. Apart from a few more operators, which will not be used in this thesis, μCRL also defines two special actions, which we have already met earlier in this chapter: τ for internal actions and δ for deadlocks.

Processes and Linear Process Equations (LPEs) In μCRL, system behavior is defined by processes. Every process defines a control flow of actions $\iota \in \mathcal{A}$, which either happen completely concurrently, or which are synchronized by communication. In the linearized form of μCRL, LPEs, which we will regard here for reasons of convenience, these processes are combined to a single one.

An LPE in μCRL in general is of the following form (Usenko, 2002, we restrain for convenience to single data parameters only):

$$
\underbrace{X(\overbrace{d : \mathbb{S}}^{\text{global vars}})}_{\text{process}} = \underbrace{\sum_{i \in I}}_{\text{summands}} \underbrace{\sum_{e_i : \mathbb{S}_i}}_{\text{local vars}} \underbrace{s_i(f_i(d, e_i))}_{\text{action call}} \cdot \underbrace{X(h_i(d, e_i))}_{\text{recursive call}} \lhd \underbrace{g_i(d, e_i)}_{\text{guard}} \rhd \delta
$$

The above formula reads as follows: $X(d : \mathbb{S})$ describes a process X with a single process variable d of sort $\mathbb{S}$. This variable d is global over the whole process. The process X can in any state make a nondeterministic choice between several actions

s_i. This choice is made over *summands* $i \in I$, with I being the set of all summands for process X.

For each single summand in our examplary specification, there exists a variable e_i of sort $\mathbb{S}_i$, which is local for that particular summand. A local variable is declared using a summation operator (over data this time). Within the scope of a summand, both the process-global variables (d in our case) and local variables (e_i) are visible.

Apart from the declaration of data variables, a summand is always of the form $\iota \triangleleft g \triangleright \iota'$. If the guard g, in our exemplary specification g_i with two parameters, evaluates to $\top$, then the action ι (in our example $s_i(f_i(d, e_i))$), otherwise the action ι' is executed. Instead of a single action, sequences of actions can be invoked; this also includes (recursive) calls to processes (in our example a call to X with a changed parameter computed by $h_i(d, e_i)$). In the linearized form of μCRL specifications, ι' is always δ. If the guard thus evaluates to $\bot$, another summand must be able to fire in the system, or the system completely deadlocks.

As a final remark, the occurrence of $f_i(d, e_i)$ as a parameter for s_i shall be explained. The function f_i is defined by rewrite rules, as we have seen it in the previous subsection. When the actual summand fires, $f_i(d, e_i)$ is rewritten to a single data value, using the actual values for d and e_i as parameters for f_i. The result of rewriting is then used as a parameter to s_i.

The LPE for a system is initialized with the line

$$\textbf{init } X(d_{init})$$

with $d_{init} \in \mathbb{S}$ an actual value for the process variables.

Relation between μCRL and IOSTSs With μCRL we can specify systems as we can with the IOSTSs defined in Definition 2.5. In fact, an IOSTS can serve as a graphical representation of a linearized μCRL specification. In the remainder of this section, we want to give a short, more informal insight into the relation between these two concepts.

In a linearized μCRL specification, we have exactly one process. In this process, the locations of the corresponding IOSTS (set L) can be stored in a process-global variable. The summands of this process correspond directly to edges:

$$\text{OUT: } \frac{X(\ell, d : \mathbb{S}) \text{ has summand } s(e).X(\hat{\ell}, d') \triangleleft g \triangleright \delta}{\ell \xrightarrow{g \triangleright ! s(e)} \hat{\ell}, \eta' = \eta_{[d \mapsto d']}}$$

$$\text{IN: } \frac{X(\ell, d : \mathbb{S}) \text{ has summand } \sum_{x:\mathbb{S}_x} s(x).X(\hat{\ell}, d') \triangleleft g \triangleright \delta}{\ell \xrightarrow{g \triangleright ? s(x)} \hat{\ell}, \eta' = \eta_{[d \mapsto d']}}$$

A distinction between input and output actions on the level of μCRL is not directly possible; this distinction must be introduced at a later transformation step.

2.5 Specifications and Abstraction

As defined in Definition 2.5, the control and data flow within a system is defined by locations (control flow) and valuations (data flow). These are combined to pairs (ℓ, η), which form the states of a system. In an STS, these states are handled symbolically, i.e. they contain variables with either constant values assigned or with a range of possible values.

When we now transform this STS into its corresponding LTS, we have to enumerate over all values of a variable rather than treating the variable symbolically. In the course of this enumeration, the state space grows and might grow over all limits – an effect known as *state (space) explosion*.

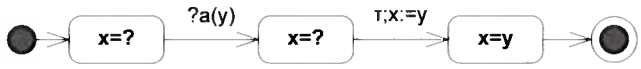

Figure 2.5: The specification of a simple system

As an example for state space explosion, let us consider the system from Figure 2.5. The system has one internal variable, x, and receives an event $a(y)$ from its environment. Subsequently, y is assigned to x. Let now $x, y \in \mathbb{N}$, i.e. both variables have an infinite domain. Since they are both unbound, enumerating all values for both variables leads to the LTS in Figure 2.6, which has not three states, but infinitely many. The state space has exploded.

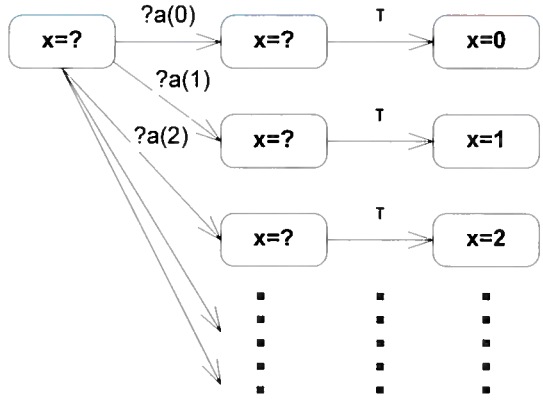

Figure 2.6: The LTS of the simple system

Analyzing such an infinitely big system to either generate appropriate test cases or to perform model checking is not feasible, neither for reasons of memory consumption nor for the duration of the analysis. Abstraction is a technique to make such systems accessible for enumerative analysis approaches. By using abstraction, the *concrete states* of a system are collected to *abstract states*. Let us consider the concrete system

from Figure 2.6 again. It has not only an infinite number of outgoing transitions from its first state, but consequently also an infinite number of subsequent states. Let us now perform an abstraction, by which we divide the interval of the natural numbers into the two classes "< 2", denoted by constant c_1, and "$\geqslant 2$", denoted by c_2. In the consequence, we retrieve the abstract system from Figure 2.7, which has only five abstract states left. These five states can then easily be analyzed.

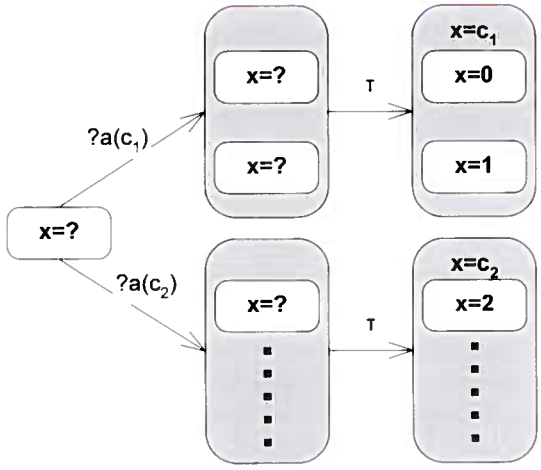

Figure 2.7: The LTS of the abstracted simple system

The application of abstraction to systems goes back to the work on Abstract Interpretation by Cousot and Cousot (1977). Abstraction techniques can coarsely be divided into two categories: approaches, which are based on Galois connections (Erné et al., 1993; Loiseaux et al., 1995), and those based on homomorphisms. In general, it can be stated, that abstractions based on galois connections are more flexible, since they do not only come with an abstraction function, but also with a concretization function. This allows us to restore the concrete system from the abstracted one. Homomorphic abstractions just provide an abstraction function.

Since we are not interested in this latter fact, we will in this thesis refer only to abstractions which are based on homomorphisms. These techniques define a homomorphic relation between a system $\mathfrak{S}$ or $\mathfrak{M}$, and its abstraction $\mathfrak{S}^\alpha$ or $\mathfrak{M}^\alpha$, resp. An abstraction function is normally *not an injective* one, so that in most cases the original system cannot be restored from the abstracted one by applying an inverse of the abstraction function. For our cases, this is completely sufficient, since we will use the abstract system to apply enumerative methods on, and the concrete system together with the symbolic technique of constraint solving.

In this thesis, we use abstraction on the level of system specifications, that is on the level of STSs, to gain a *safe* abstraction of a system. A safe abstraction preserves all behavior from the concrete system also in the abstract system. This can be achieved by choosing an abstraction, which overapproximates the concrete system, i.e. which

adds behavior. In order to do so, we abstract data, for instance action parameters, by combining them into equivalence classes as shown in Figure 2.7.

Such an abstraction is, in its extreme form, the *chaotic data abstraction* proposed by Sidorova and Steffen (2001b), which unites all possible input data of a system to a single class $\mathbb{T}$. We will describe this abstraction technique in more detail in Section 4.3. Abstracting systems on the level of STSs rather than that of LTSs inherently leads to a further overapproximation on the level of the abstracted systems, as has been shown by Clarke et al. (1994). Since we are interested in overapproximations anyway, this fact does not affect our further work.

Chapter 3

Transformation from μCRL to Prolog

> The Babel fish is small, yellow and leechlike, and probably the oddest thing in universe. [...] The practical upshot of all this is that if you stick a Babel fish in your ear you can instantly understand anything said to you in any form of language.
>
> *(Douglas Adams)*

Reactive systems with data, as we regard them in this thesis, build up a system-internal web of interdependencies between different variables. These interdependencies affect the relation between input and output data on the one hand, on the other hand they also determine the possible order of events, which the system exchanges with its environment or accepts from it. When examining such a system, either by testing it or checking its models, it is necessary to make this internal, quite implicit knowledge explicit in order to receive sustainable validation and verification results.

In model checking, different variables in a system are set into relation with each other by either an explicit enumeration over all possible combinations of variable values – a tedious approach which is often even doomed to failure – or by symbolic approaches for model checking. In testing, this is the task of the so-called *oracle* (or test oracle) which predicts the expected results for a particular test run.

In testing, the oracle is not necessarily an automated solution, since in many cases the prediction must be based on an informal model and is thus coded directly into the test cases by a human being. However, model-based test approaches in a model-driven software development approach with models of a certain maturity available can do without the human and automate the test oracle as well. The state of the art in model checking has reached this point anyway, since the models regarded in this area are formal and provide thus the maturity necessary for fully automatic symbolic model checking.

A suitable class of systems for computations on interdependent data elements for symbolic model checking and in test oracles are constraint solvers. Constraint solvers interpret so-called Constraint Logic Programs (CLPs), which themselves encode data elements and their interdependencies. By solving a CLP, the solver constrains the possible values for each of the data elements to those sets, for which a solution is possible. A symbolic model checker or a tester can base its further actions on the results retrieved from the constraint solver.

To be able to apply constraint solving on a system specification given in μCRL, as in our case, this specification must first be translated into a CLP in Prolog. We have

developed a transformation of symbolic system specifications in μCRL into CLPs for ECLiPSe Prolog. On the one hand, we designed a transformation of data equations to rules in Prolog simulating term rewriting, on the other hand we also created an accordant transformation of process equations to Prolog as well.

In this chapter, we will first give a general introduction to constraint solving in the following section. Afterwards, we will present an approach, which we developed for the transformation of formal specifications in μCRL to CLPs which can be processed by the constraint solver of ECLiPSe Prolog. In Section 3.2, we will first discuss the transformation of ADT and their equation systems to Prolog. This transformation is based on the idea to simulate term rewriting, as it normally takes place when μCRL specifications are processed, within a Logic Program (LP) without the particularities of constraint solving. We will prove the correctness of our approach and then extend it for some aspects, where actual constraint solving shows its strengths best, namely for arithmetics in combination with numerical and symbolic data types. In Section 3.3, we will turn our focus to the behavioral part of a μCRL specification and describe our approach for the transformation of guarded transitions in an IOSTS or a μCRL specification to Prolog. Finally, in Section 3.4,we will compare our approach of the transformation of formal specifications to CLPs to one developed by Pretschner et al. (2004a,b).

3.1 Constraint Solving

Variables in a system or in a single trace of a system's control and data flow are often restricted in their values and value domains. These restrictions are in many cases induced by interdependencies of variables and are enforced by conditions (or guards), in the system. We employ constraint solving to find suitable values and value domains for the affected variables in this system.

In order to do so, we use the declarative programming language Prolog with constraint solving extensions. In a declarative programming language, the problem to be solved is itself defined, rather than a particular solution for it. This definition takes place in a LP. This program is then executed by asking a query to it and, in the optimal case, produces a solution to the stated problem.

In this subsection, we will first introduce LPs in pure Prolog as a basis for CLPs. Then, in a second part, we introduce the notion of constraint nets (Guesgen, 2000), as an intuitive access to the problem of constraint solving, and discuss the syntax of constraints.

Logic Programming with Prolog

Regarding Prolog, we first have to distinguish two areas, for which this language is used: logic programming with pure Prolog (Clocksin and Mellish, 1994) as well as constraint logic programming with additional constraint solver libraries (Apt and

Wallace, 2007). In this part, we will define a boolean algebra and an algebra of natural numbers for Prolog.

The boolean algebra for Prolog defines a standard boolean algebra with $\top$ and $\bot$ as well as the operations $\neg$, $\wedge$, $\vee$ and equivalence with the syntax, i.e. the signature, of Prolog.

Definition 3.1 (Signature for booleans in Prolog). The signature for booleans in Prolog is defined as

$$\mathfrak{Z}_{\mathrm{Bool_p}} = (\mathrm{Bool_p}, \{\Box_p, \mathrm{fail_p}, =_p, \wedge_p, \vee_p, \mathrm{not_p}\}, \sigma_{\mathrm{Bool_p}})$$

with the signature of functions $\sigma_{\mathrm{Bool_p}}$ being defined as

$$\begin{aligned}
\Box_p &: \rightarrow \mathrm{Bool_p} \\
\mathrm{fail_p} &: \rightarrow \mathrm{Bool_p} \\
=_p &: \mathrm{Bool_p} \times \mathrm{Bool_p} \rightarrow \mathrm{Bool_p} \\
\wedge_p &: \mathrm{Bool_p} \times \mathrm{Bool_p} \rightarrow \mathrm{Bool_p} \\
\vee_p &: \mathrm{Bool_p} \times \mathrm{Bool_p} \rightarrow \mathrm{Bool_p} \\
\mathrm{not_p} &: \mathrm{Bool_p} \rightarrow \mathrm{Bool_p}
\end{aligned}$$

■

Syntactically, the term $\wedge_p$ is in Prolog expressed as a comma (,), while the $\vee_p$ is expressed as a semicolon (;). The Prolog predicate $\mathrm{fail_p}$ is syntactically expressed as `fail`, while $\Box_p$ (or short: $\Box$) has no explicit syntactic counterpart, but leaving a clause empty.

Now, we define an algebra for natural numbers in Prolog. Unlike the algebra for μCRL in Example 2.21, the standard operations are accompanied by constants $\underline{c_i}$, effectively *numbers*, here, rather than being encoded by successor terms.

Definition 3.2 (Signature for natural numbers in Prolog). The signature for natural numbers in Prolog is defined as

$$\mathfrak{Z}_{\mathrm{Nat_p}} = (\mathrm{Nat_p} \cup \mathrm{Bool_p}, \{\{\underline{c_i} | c_i \in \mathbb{N}\}, +_p, \cdot_p, =_p, >_p, \geqslant_p, <_p, \leqslant_p\}, \sigma_{\mathrm{Nat_p}})$$

with the signature of functions $\sigma_{\mathrm{Nat_p}}$ being defined as

$$\begin{aligned}
\underline{c_i} &: \rightarrow \mathrm{Nat_p} \\
+_p &: \mathrm{Nat_p} \times \mathrm{Nat_p} \rightarrow \mathrm{Nat_p} \\
\cdot_p &: \mathrm{Nat_p} \times \mathrm{Nat_p} \rightarrow \mathrm{Nat_p} \\
=_p &: \mathrm{Nat_p} \times \mathrm{Nat_p} \rightarrow \mathrm{Bool_p} \\
>_p &: \mathrm{Nat_p} \times \mathrm{Nat_p} \rightarrow \mathrm{Bool_p} \\
\geqslant_p &: \mathrm{Nat_p} \times \mathrm{Nat_p} \rightarrow \mathrm{Bool_p} \\
<_p &: \mathrm{Nat_p} \times \mathrm{Nat_p} \rightarrow \mathrm{Bool_p} \\
\leqslant_p &: \mathrm{Nat_p} \times \mathrm{Nat_p} \rightarrow \mathrm{Bool_p}
\end{aligned}$$

■

In addition to the above-mentioned operators $\wedge_p$, $\vee_p$ and not_p, Prolog also provides an operator for *if-then-else* constructs.

Definition 3.3 (*If-then-else* in Prolog). Let $\Im_{Bool_p} = (Bool_p, \mathcal{F}_{Bool_p}, \sigma_{Bool_p})$ be the signature of the boolean algebra for Prolog. Let furthermore $t_1, t_2, t_3 \in \mathcal{T}_{Bool_p}(\mathcal{F}_{Bool_p}, X)$ be terms in Prolog. Then: $t_1 \rightarrow t_2; t_3 \equiv (t_1 \wedge_p t_2) \vee_p (not_p(t_1) \wedge_p t_3)$. ■

LPs in Prolog are defined by a set of *rules*, which formulate statements about objects and the relations between objects. We can distinguish rules in general and tautological rules, which are known as *facts*. Furthermore, we can ask *queries* to Prolog, which are solved using the predefined set of rules in the LP.

Definition 3.4 (Rule). Let f be a function symbol and $t_1, \ldots, t_n$ be terms of any sort. Then a rule in Prolog is defined as a tuple $\rho = (f(t_1, \ldots, t_n), q)$, denoted

$$f(t_1, \ldots, t_n) \leftarrow q,$$

with $f : \ldots \rightarrow Bool$, $q :\rightarrow Bool$. The left side, the head, consists of a predicate (f) and a list of arguments, the right side consists of a query q. ■

Definition 3.5 (Fact). A *fact* is a rule $\rho = (f(t_1, \ldots, t_n), \Box)$. ■

Definition 3.6 (Query). A *query* is a rule $\rho = (\Box, q)$. ■

A procedure in Prolog is a collection of rules with the same predicate. A Logic Program (LP) consists of a set of procedures.

Queries to LPs are in Prolog solved following the *resolution principle* introduced by Robinson (1965) for *Horn clauses*. Our definitions for rules, facts and queries correspond to that of Horn clauses (Clocksin and Mellish, 1994). Let us consider a query as a conjunction[1]

$$\rho_q^0 = (\Box, q_1 \wedge \ldots \wedge q_n)$$

and an LP

$$\begin{aligned} \mathfrak{P} = \{ \quad & \rho_1 \leftarrow q_{1_1} \wedge \ldots \wedge q_{1_m}, \\ & \vdots \\ & \rho_n \leftarrow q_{n_1} \wedge \ldots \wedge q_{n_o}, \\ & \vdots \\ & \ldots\}. \end{aligned}$$

Prolog starts resolving the query by trying to find a matching rule for q_1 in $\mathfrak{P}$. Assume, there is a unification θ_1 for variables in q_1 and ρ_1, such that $q_1^{\theta_1} = \rho_1^{\theta_1}$. Then q_1 is unified in one resolution step, such that the complete query is transformed to

$$\rho_q^1 = (\Box, q_{1_1}^{\theta_1} \wedge \ldots \wedge q_{1_m}^{\theta_1} \wedge q_2^{\theta_1} \wedge \ldots \wedge q_n^{\theta_1}).$$

[1] Disjunctions are possible, but can be expressed as adjacent queries, so we do not want to regard those here.

Now Prolog tries to find a resolvent for $q_{1_1}^{\theta_1}$ to unify it with. Unifying a query with a fact has the consequence, that the query under consideration is dropped out, and Prolog subsequently tries to find a resolvent for the following query. Regarding the query above as

$$\rho_q^1 = (\square, q_{1_1}^{\theta_1} \wedge q_{1_2}^{\theta_1} \wedge \ldots \wedge q_{1_m}^{\theta_1} \wedge \ldots \wedge q_{n}^{\theta_1})$$

and assuming a rule $\rho_{1_1} \in \mathfrak{P}$ with $\rho_{1_1}^{\theta_2} \leftarrow \square$, $\rho_{1_1}^{\theta_2} = q_{1_1}^{\theta_1\theta_2}$. Then, the resulting query is reduced to

$$\rho_q^2 = (\square, q_{1_2}^{\theta_1\theta_2} \wedge \ldots \wedge q_{1_m}^{\theta_1\theta_2} \wedge \ldots \wedge q_{n}^{\theta_1\theta_2})$$

and Prolog goes on treating $q_{1_2}^{\theta_1\theta_2}$. The query succeeds, if it can be resolved to $\rho_q^k = (\square, \square)$, $k \in \mathbb{N}$.

While for a query and rules without any variables, a rule matches a query if they are both *identical*, this is not the case when we introduce variables into our LP. Matching queries and rules with variables means finding a combination of both, where the predicate is identical and variables *can be instantiated* in a way, that they also get identical values. If such an instantiation does not exist, the rule does not match the query. If such an instantiation does exist, variables for which this instantiation is unique get instantiated, while those for which there is no *unique* instantiation stay uninstantiated. Furthermore, the value of an instantiated variable is propagated to all equally named variables in a conjunction of queries (either the respective rule or ρ_q).

Definition 3.7 (Relation $\Rightarrow_{LP}$). We write $\rho_q \Rightarrow_{LP} \rho_q'$ if there is a resolution of ρ_q which leads after one or more steps to ρ_q'. ■

A rule in Prolog does not have a return value as in other, for instance imperative, programming languages. A possible return value is defined as one more data element in the argument list of ρ. Another limitation of Prolog is its missing type system. Variables in Prolog are untyped, which means that we have to introduce the simulation of a type system in order to achieve typedness of Prolog LPs.

Constraint Logic Programming

Constraint logic programming extends logic programming by the idea of constraints. Constraints define the interrelation between different data elements like variables.

Definition 3.8 (Constraint; Guesgen, 2000). A *constraint* $\mathfrak{C}$ is a pair $\mathfrak{C} = (X, \mathfrak{R})$ of a set of variables $X = \{x_1, \ldots, x_n\}$ with domains $\mathbb{D}_1, \ldots, \mathbb{D}_n$ and a decidable relation $\mathfrak{R} \subseteq \mathbb{D}_1 \times \ldots \times \mathbb{D}_n$ between these variables. ■

Examples for constraints are $(\{x\}, \leqslant)$, defining $x \leqslant 5$ or $\leqslant (x, 5)$, resp., or $(\{x, y\}, f)$ with $f : \mathbb{N} \times \mathbb{N}$ being defined as $f(x, y) \Leftrightarrow x \leqslant 5 \wedge y > x$.

Different constraints on variables can form a so-called *constraint net* by which a number of variables is interconnected with each other by a number of constraints.

Definition 3.9 (Constraint net; ibid). A *constraint net* on the variables $x_1, \ldots, x_n$ is a set of constraints, s.t. all variables of each constraint are a subset of $x_1, \ldots, x_n$. ■

A CLP defines such a constraint net. A *constraint net* is also named a *constraint domain* $\mathfrak{D}$ (Marriott and Stuckey, 1998). A *logical theory* $\mathfrak{T}$ determines, which constraints in $\mathfrak{D}$ hold under a certain valuation θ, and which do not. If a constraint $\mathfrak{C}$ holds under $\mathfrak{T}$ of a constraint domain $\mathfrak{D}$, this is denoted $\mathfrak{D} \models [\![\mathfrak{C}]\!]_\theta$. In this case, θ is a *solution* for $\mathfrak{C}$. Such a solution can be found only for a k-consistent constraint net with k the number of variables.

Definition 3.10 (k-Consistency; ibid). Let there be a constraint net over the variables $x_1, \ldots, x_n$ of domains $\mathbb{D}_1, \ldots, \mathbb{D}_n$. Let $(t_{i_1}, \ldots, t_{i_{k-1}}) \in \mathbb{D}_{i_1} \times \ldots \times \mathbb{D}_{i_{k-1}}$ with $i_j = \{1, \ldots, n\}$ for $j = \{1, \ldots, k\}$ be the assignment of values to $k-1$ pair-wise different variables, which fulfills all constraints between these variables. The net is k-consistent, if, by adding an arbitrary variable x_{i_k} with $k \in \{1, \ldots, n\}$, the assignment of values can be extended to $(t_{i_1}, \ldots, t_{i_k}) \in \mathbb{D}_{i_1} \times \ldots \times \mathbb{D}_{i_k}$ such that all constraints between these k variables are also fulfilled. ■

Definition 3.11 (Consistency). A constraint net over the variables $x_1, \ldots, x_n$ is *consistent*, if it is k-consistent for $k = n$. ■

Example 3.12. In Figure 3.1, an exemplary net of binary constraints over the variables x, y, z is shown. All variables initially (0) have the domain $\mathbb{D} = \{1; 2; 3; 4; 5\}$. The constraints over the variables are $x < y$, $x < z$ and $z < y$. We now stepwise make this constraint net consistent (i.e. 3-consistent for this net), s.t. the data domains of each of the variables reflects the possible values, which can be assigned to the respective variable without violating any constraints. In a first step (1), we regard the constraint $x < y$. Since neither $x = 5$ nor $y = 1$ would satisfy this constraint, we remove them from the respective domains. Then (2), we regard $x < z$ and remove $z = 1$ for the same reason. Afterwards (3), we remove $y = 2$ and $z = 5$ regarding $z < y$. Checking the consistency of the net, we find out, that the last operation did not affect the constraint $x < y$, but leads to an inconsistency regarding the previously solved constraint $x < z$, s.t. we have to remove $x = 4$ in a last step (4). Now, the whole constraint net is consistent.

From a consistent constraint net, we can now select a solution θ. For our example, such a solution is $\theta = \{x \mapsto 2, y \mapsto 4, z \mapsto 5\}$. However, not every combination of values in a consistent constraint net can be a solution; $\{x \mapsto 3, y \mapsto 2, z \mapsto 3\}$, for instance, violates the constraints $x < y$ and $x < z$ and is thus no solution for the set of constraints.

If the domain of one or more variables becomes empty while solving a constraint net, then there exists no solution for the set of constraints. Furthermore, it should be mentioned, that constraints do not always have to be binary, as shown in the example. Already relatively simple expressions like $x = y + z$ result in n-ary relations. This holds a fortiori for complicated sets of constraints, like those over a trace in a system.

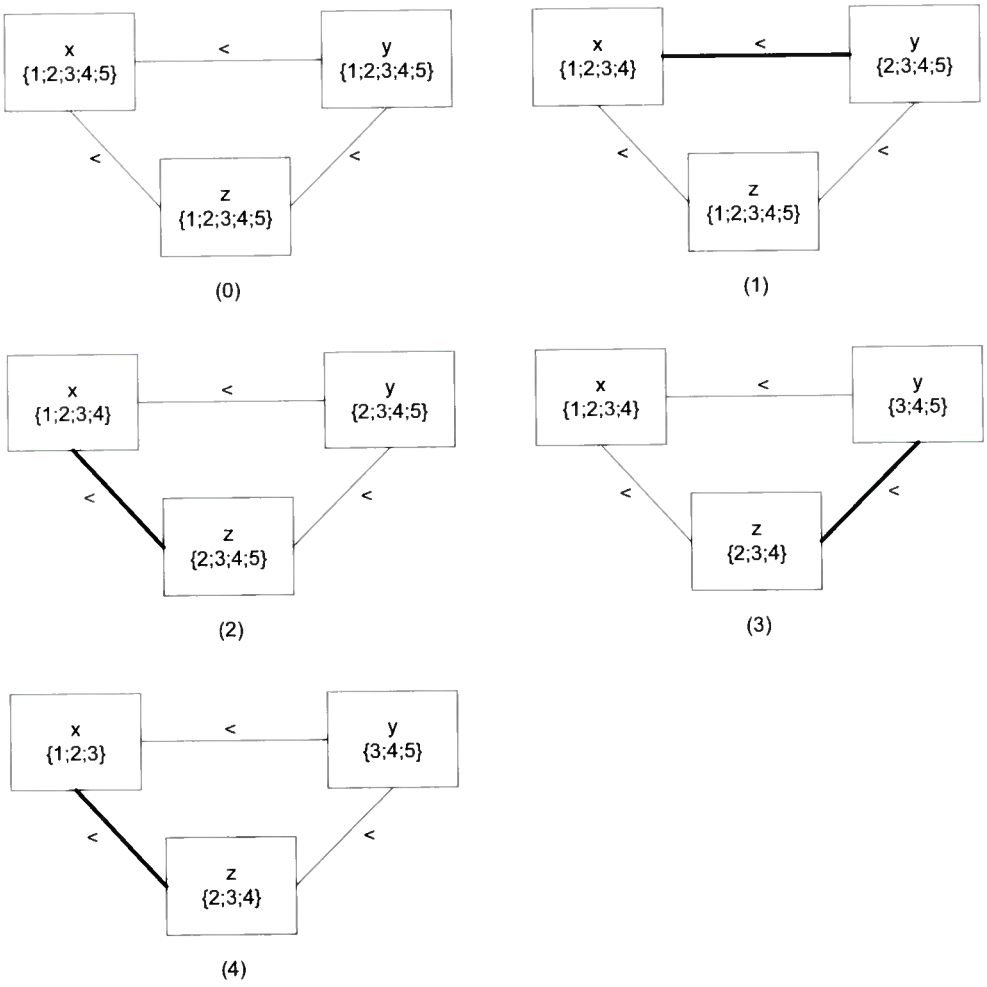

Figure 3.1: A sample constraint net

3.2 Transformation of Abstract Datatypes to Prolog

In the following two sections, we develop a transformation of μCRL specifications to Prolog CLPs. By that, we are able to apply constraint solving on data selection for systems specified in μCRL. To give the reader an idea of this transformation, we begin with a small example. It is the specification of a sort for lists in μCRL with only one operation which provides the length of a given list. The associated process defines an infinite loop which performs an action notempty, if the list is not empty, i.e. its length is greater than 0. The system is specified as follows (in excerpts; the definitions of Bool and Nat are standard definitions as given in the previous chapter):

$$
\begin{array}{ll}
\textbf{sort} & \mathrm{List}, \mathrm{D}, \mathrm{Bool}, \mathrm{Nat} \\
\textbf{func} & \% \text{ definitions of Bool and Nat} \\
 & \ldots \\
 & \mathrm{empty} :\rightarrow \mathrm{List} \quad \mathrm{list} : \mathrm{D} \times \mathrm{List} \rightarrow \mathrm{List} \quad \mathrm{d} :\rightarrow \mathrm{D} \\
\textbf{map} & \mathrm{gt} : \mathrm{Nat} \times \mathrm{Nat} \rightarrow \mathrm{Bool} \\
 & \mathrm{len} : \mathrm{List} \rightarrow \mathrm{Nat} \\
 & \ldots \\
\textbf{var} & \mathrm{l} : \mathrm{List}, \mathrm{x} : \mathrm{D}, \mathrm{y}_1, \mathrm{y}_2 : \mathrm{Nat} \\
\textbf{rew} & \mathrm{gt}(0,0) = \mathrm{F} \quad \mathrm{gt}(0, \mathrm{S}(\mathrm{y}_2)) = \mathrm{F} \quad \mathrm{gt}(\mathrm{S}(\mathrm{y}_1), 0) = \mathrm{T} \\
 & \mathrm{gt}(\mathrm{S}(\mathrm{y}_1), \mathrm{S}(\mathrm{y}_2)) = \mathrm{gt}(\mathrm{y}_1, \mathrm{y}_2) \\
 & \mathrm{len}(\mathrm{empty}) = 0 \quad \mathrm{len}(\mathrm{list}(\mathrm{d}, \mathrm{l})) = \mathrm{S}(\mathrm{l}) \\
\textbf{act} & \mathrm{notempty} \\
\textbf{proc} & \mathrm{X}(\mathrm{l} : \mathrm{List}) = \mathrm{notempty}.\mathrm{X}(\mathrm{l}) \lhd \mathrm{gt}(\mathrm{len}(\mathrm{l}), 0) \rhd \delta
\end{array}
$$

The above specification is now transformed according to our approach to the following Prolog CLP:

$$
\begin{array}{l}
\mathrm{gt}(\mathbb{N}(\mathrm{x}), \mathbb{N}(\mathrm{y}), \mathbb{B}(\mathrm{r})) \leftarrow (\mathrm{x} \#> \mathrm{y} \rightarrow \mathrm{r} = \top; \mathrm{r} = \bot) \\
\quad \ldots \\
\mathrm{len}(\mathbb{L}(\mathrm{empty}), \mathbb{N}(\mathrm{x}_1)) \leftarrow \mathrm{x}_1 =_{\mathbb{N}} 0 \\
\mathrm{len}(\mathbb{L}(\mathrm{list}(\mathbb{D}(\mathrm{d}), \mathbb{L}(\mathrm{y}))), \mathbb{N}(\mathrm{x}_1)) \\
\qquad \leftarrow \mathrm{len}(\mathbb{L}(\mathrm{y}), \mathbb{N}(\mathrm{x}')) \wedge_{\mathrm{p}} \mathrm{x}'' =_{\mathbb{N}} 1 \wedge_{\mathrm{p}} \\
\qquad \mathrm{x}_1 =_{\mathbb{N}} \mathrm{x}_2 \wedge_{\mathrm{p}} \mathrm{add}(\mathbb{N}(\mathrm{x}'), \mathbb{N}(\mathrm{x}''), \mathbb{N}(\mathrm{x}_2)) \\
\quad \ldots \\
\mathrm{notempty}(\mathrm{global}(\mathbb{L}(\mathrm{l})), \mathrm{global}(\mathbb{L}(\mathrm{l})), \mathrm{param}) \\
\qquad \leftarrow \mathrm{len}(\mathbb{L}(\mathrm{l}), \mathbb{N}(\mathrm{x})), \mathrm{gt}(\mathbb{N}(\mathrm{x}), \mathbb{N}(0), \mathbb{B}(\mathrm{b})), \mathbb{B}(\mathrm{b})
\end{array}
$$

3.2.1 Syntactical Transformation of μCRL ADTs to Prolog

In the previous section, we introduced constraint solving as a tool to solve the interdependencies of variables in a system in general. In the remainder of this thesis, we will apply constraint solving on the data of a system specified in μCRL. Therefor we have to transform the several parts of a μCRL specification to a CLP in Prolog. In this section, we will discuss our approach to transform ADTs from μCRL to Prolog. In order to do so, we will describe the transformation of terms to pure Prolog and prove it equivalent to a fragment of term rewriting Bergstra et al. (1989). First of all, we define the general structure of CLPs for specifications.

Definition 3.13 (CLP $\mathfrak{P}(\mathfrak{S})$). The CLP for a specification $\mathfrak{S}$ is a tuple $\mathfrak{P}(\mathfrak{S}) = (\mathfrak{P}_{\text{Adt}}(\mathfrak{S}), \mathfrak{P}_{\text{Proc}}(\mathfrak{S}))$ consisting of two sets of Prolog rules with

$\mathfrak{P}_{\text{Adt}}(\mathfrak{S})$ representing the equations $\varepsilon \in \mathcal{E}$ in $\mathfrak{S} = (\mathcal{Z}, \mathcal{E})$ and

$\mathfrak{P}_{\text{Proc}}(\mathfrak{S})$ representing the summand rules from Proc in $\mathfrak{S}$.

■

In the remainder, we will abbreviate $\mathfrak{P}(\mathfrak{S})$ by $\mathfrak{P}$. Furthermore, we will concentrate on $\mathfrak{P}_{\text{Adt}}$ only, since we consider ADTs. We will now first regard the treatment of constructors in general and for numerical data, before later discussing the transformation of equations.

We will first need to introduce the notion of *stacks*, which will be used later in order to define the transformation functions for terms to Prolog. A stack is a last-in first-out memory with the two operations push and pop.

Definition 3.14 (Stack). Let a stack be defined for data elements of type T as follows:

sort	$\mathrm{Stack}, \mathrm{T}, \mathrm{Bool}$
func	$[] :\rightarrow \mathrm{Stack}$
	$\mathrm{push} : \mathrm{T} \times \mathrm{Stack} \rightarrow \mathrm{Stack}$
map	$\mathrm{pushS} : \mathrm{Stack} \times \mathrm{Stack} \rightarrow \mathrm{Stack}$
	$\mathrm{pop} : \mathrm{Stack} \rightarrow \mathrm{T} \times \mathrm{Stack}$
	$\mathrm{empty} : \mathrm{Stack} \rightarrow \mathrm{Bool}$
var	$x : \mathrm{T},\ s_1, s_2 : \mathrm{Stack}$
rew	$\mathrm{pushS}(s_1, []) = s_1$
	$\mathrm{pushS}([], s_2) = s_2$
	$\mathrm{pushS}(\mathrm{push}(x, s_1), s_2) = \mathrm{push}(x, \mathrm{pushS}(s_1, s_2))$
	$\mathrm{pop}(\mathrm{push}(x, s_1)) = (x, s_1)$
	$\mathrm{empty}([]) = \mathrm{T}$
	$\mathrm{empty}(\mathrm{push}(x, s_1)) = \mathrm{F}$

■

Terms are transformed by the functions transformTerm and transformTerm′. A term in μCRL is provided to transformTerm for processing. This function passes the term on to transformTerm′. Here, all parameters of the term are first processed recursively. The result is a stack, which contains variable assignments and Prolog terms for the processed parameters. If now the main term processed by transformTerm′ is a constructor term, then the stack with necessary variable assignments for the term's parameters is returned together with the term itself as a Prolog term. If the main term is a function term, then it is processed and added itself to the stack of variable assignments. The second component of the result of transformTerm′ is then the variable, to which the result of the function term is assigned to. As a last transformation step, transformTerm constructs a conjunction of all variable assignments from the stack and the processed main term. The two functions are defined below.

Definition 3.15 (Transforming terms). Let x_i be fresh variables and t_i' be fresh terms. For each sort $\mathbb{S}_i$ with $t_i : \mathbb{S}_i$, we introduce a function symbol $\mathbb{D}_i : \mathfrak{T} \to \mathfrak{T}$ with $\mathbb{D}_i$ the domain denoted by $\mathbb{S}_i$ for explicit typing in Prolog. Furthermore, we introduce an operator $=_{\mathbb{D}_i} : \mathbb{D}_i \times \mathbb{D}_i \to \mathbb{B}$ for each domain of t_i. Then, the helper function

$$\mathrm{transformTerm}' : \mathfrak{T}(\mathcal{F}, X) \to \mathrm{Stack}[\mathfrak{T}(\mathcal{F} \cup \mathcal{S} \cup \{=_{\mathbb{D}}\}, X \cup X')] \times \mathfrak{T}(\mathcal{F} \cup \mathcal{S}, X \cup X')$$

is defined in Algorithm 3.1. The function

$$\mathrm{transformTerm} : \mathfrak{T}(\mathcal{F}, X) \to \mathfrak{T}(\mathcal{F} \cup \mathcal{S} \cup \mathrm{Bool}_p \cup \{=_{\mathbb{D}}\}, X \cup X')$$

itself is defined in Algorithm 3.2. ■

The algorithm takes an arbitrary term $f(t_1, \ldots, t_n)$ and initializes a stack to cache subterms. In case, this term is a function term, it transforms all parameters t_i to t_i', finally pushes the Prolog rule for f with the parameters t_i' and an additional parameter for the function return value on the stack and adds this additional parameter as the second element of the result pair of the algorithm. In case, the processed term is a constructor term, its parameters are processed in an equal way, but the whole transformed term is not pushed on the stack, but forms the second element of the result pair of the algorithm. In this case, there is no fresh variable introduced, either.

This algorithm transforms a term with transformTerm′. The elements of the resulting stack are afterwards conjoined by the operator $\wedge_p$ with the top element of the stack being the last conjunct. All other elements of the stack keep their order. If the transformed term is not a function term, a new variable is introduced and the term is assigned to it.

Example 3.16. Let $\mathrm{list}, \mathrm{empty}, d_1, d_2 \in C$ be constructors, of which list with an arity of 1 and empty, d_1, d_2 with an arity of 0. Let furthermore $\mathrm{len} \in M$ be a function. Then:

$$\mathrm{transformTerm}(\mathrm{list}(d_1, \mathrm{list}(d_2, \mathrm{empty}))) \twoheadrightarrow x =_{\mathbb{L}} \mathrm{list}(\mathbb{D}(d_1), \mathbb{L}(\mathrm{list}(\mathbb{D}(d_2), \mathbb{L}(\mathrm{empty}))))$$

Algorithm 3.1 transformTerm′

Require: $f(t_1, \ldots, t_n)$
Ensure: $pair \in Stack \times \mathfrak{T}$
1 $L' := []$; x is fresh variable;
2 **if** $f \in M$ **then**
3 **for all** $1 \leqslant i \leqslant n$ **do**
4 $(L_i, t_i') := transformTerm'(t_i)$;
5 $L' := pushS(L_i, L')$;
6 **od**
7 $L := push(f(\mathbb{D}_1(t_1'), \ldots, \mathbb{D}_n(t_n'), \mathbb{D}(x)), L')$;
8 $pair := (L, x)$;
9 **else if** $f \in C$ **then**
10 **for all** $1 \leqslant i \leqslant n$ **do**
11 $(L_i, t_i') := transformTerm'(t_i)$;
12 $L' := pushS(L_i, L')$;
13 **od**
14 $pair := (L, f(\mathbb{D}_1(t_1'), \ldots, \mathbb{D}_n(t_n')))$;
15 **fi**
16 return(pair);

$$transformTerm'(list(d_1, list(d_2, empty))) \twoheadrightarrow ([], list(\mathbb{D}(d_1), \mathbb{L}(list(\mathbb{D}(d_2), \mathbb{L}(empty)))))$$

$$transformTerm'(d_1) \rightarrow ([], d_1)$$

$$transformTerm'(list(d_2, empty)) \twoheadrightarrow ([], list(\mathbb{D}(d_2), \mathbb{L}(empty)))$$

$$transformTerm'(d_2) \rightarrow ([], d_2)$$

$$transformTerm'(empty) \rightarrow ([], empty)$$

$$transformTerm(len(list(d_1, list(d_2, empty)))) \twoheadrightarrow len(list(\mathbb{D}(d_1), \mathbb{L}(list(\mathbb{D}(d_2), \mathbb{L}(empty)))), \mathbb{N}(x))$$

$$transformTerm'(len(list(d_1, list(d_2, empty)))) \twoheadrightarrow ([len(list(\mathbb{D}(d_1), \mathbb{L}(list(\mathbb{D}(d_2), \mathbb{L}(empty)))), \mathbb{N}(x))], x)$$

$$transformTerm'(list(d_1, list(d_2, empty))) \twoheadrightarrow ([], list(\mathbb{D}(d_1), \mathbb{L}(list(\mathbb{D}(d_2), \mathbb{L}(empty)))))$$

$$transformTerm'(d_1) \rightarrow ([], d_1)$$

$$transformTerm'(list(d_2, empty)) \twoheadrightarrow ([], list(\mathbb{D}(d_2), \mathbb{L}(empty)))$$

$$transformTerm'(d_2) \rightarrow ([], d_2)$$

$$transformTerm'(empty) \rightarrow ([], empty)$$

Algorithm 3.2 transformTerm

Require: $f(t_1, \ldots, t_n)$
Ensure: $term \in \mathfrak{T}$

```
1  if f ∈ M then
2    (B, x) := transformTerm'(f(t_1, ..., t_n));
3    (t', B) := pop(B);
4    while ¬empty(B) do
5      (b, B) := pop(B);
6      t' := b ∧_p t';
7    od
8    term := t';
9  else
10   (B, t) := transformTerm'(f(t_1, ..., t_n));
11   t' := □;
12   while ¬empty(B) do
13     (b, B) := pop(B);
14     t' := b ∧_p t';
15   od
16   x is fresh variable;
17   term := t' ∧_p x =_D t;
18 fi
19 return(term);
```

We will now define a function, which transforms algebraic equations from μCRL to Prolog. This function has a similar functionality as transformTerm from Definition 3.15. There are, however, two differences: First of all, transformEquation processes two terms at the same time, namely the left and the right hand side of an equation. Secondly, all assignments to variables concerning the term from the left hand side of an equation are after transformation of this equation to a Prolog rule defined in the *body* of the rule rather than in its head. The head of the Prolog rule contains only the main term from the left hand side of the original equation. The body of the Prolog rule contains both the variable assignments for the left hand side term, as well as the transformed right hand side term from the equation. The according function

$$\text{transformEquation} : \mathfrak{T}(\mathcal{F} \cup \{=\}, X) \to \mathfrak{T}(\mathcal{F} \cup \mathcal{S} \cup \text{Bool}_p \cup \{=_{\mathbb{D}}, \leftarrow\}, X \cup X')$$

is defined in Algorithm 3.3.

Using transformTerm', this algorithm transforms two terms t_f and t_g of an equation $t_f = t_g$. The elements of the resulting stacks are afterwards conjoined by the operator $\wedge_p$. Hereby forms the top element of the stack for t_f (B_f) the head of the rule, while all the other elements of B_f and those of the stack for t_g (B_g) are conjoined with the operator $\wedge_p$ and the top element of the stack B_g being the last conjunct. If the transformed term t_g is not a function term, it is assigned explicitly to the result

variable of the transformed term t'_f. Otherwise, the two result variables of the transformed terms t'_f and t'_g are explicitly assigned to each other just before the occurrence of the transformed main term t'_g.

Remark 3.17. The function transformEquation defined above does not reuse the function transformTerm from Definition 3.15. The reason is, that during the transformation of equations, terms have to be divided into segments, of which one part forms part of the rule head in Prolog, while the other forms part of the rule body.

We will later prove the correctness of the function transformTerm w.r.t. term rewriters with an innermost term rewriting strategy. Due to the commutativity of the $\wedge$-operator in Boolean algebras, this proof also holds for the segmented term transformation from Algorithm 3.3.

Example 3.18. As an example, let us consider two rules defining the length of a list:

$$\mathrm{len}(\mathrm{empty}) = 0 \tag{3.1}$$

$$\mathrm{len}(\mathrm{list}(x,y)) = \mathrm{S}(\mathrm{len}(y)) \tag{3.2}$$

The transformation of (3.1) happens according to the case $g \notin M$ from Algorithm 3.3:

$$\mathrm{transformEquation}((\emptyset, \mathrm{len}(\mathrm{empty}), 0)) \twoheadrightarrow \mathrm{len}(\mathbb{L}(\mathrm{empty}), \mathbb{N}(x_1)) \leftarrow x_1 =_{\mathbb{N}} 0$$

$$\mathrm{transformTerm}'(\mathrm{len}(\mathrm{empty})) \twoheadrightarrow ([\mathrm{len}(\mathbb{L}(\mathrm{empty}), \mathbb{N}(x_1))], x_1)$$
$$\mathrm{transformTerm}'(\mathrm{empty}) \rightarrow ([], \mathrm{empty})$$
$$\mathrm{transformTerm}'(0) \rightarrow ([], 0)$$

Now, we show the transformation of equation (3.2) in an LP-only setting without making use of any of ECLiPSe Prolog's CLP features.

$$\mathrm{transformEquation}((\{x,y\}, \mathrm{len}(\mathrm{list}(x,y)), \mathrm{S}(\mathrm{len}(y)))) \twoheadrightarrow \mathrm{len}(\mathbb{L}(\mathrm{list}(\mathbb{D}_x(x), \mathbb{L}(y))), \mathbb{N}(x_1)) \leftarrow \mathrm{len}(\mathbb{L}(y), \mathbb{N}(x')) \wedge_p x_1 =_{\mathbb{N}} \mathrm{S}(\mathbb{N}(x'))$$

$$\mathrm{transformTerm}'(\mathrm{len}(\mathrm{list}(x,y))) \twoheadrightarrow ([\mathrm{len}(\mathbb{L}(\mathrm{list}(\mathbb{D}_x(x), \mathbb{L}(y))), \mathbb{N}(x_1))], x_1)$$
$$\mathrm{transformTerm}'(\mathrm{list}(x,y)) \twoheadrightarrow ([], \mathrm{list}(\mathbb{D}_x(x), \mathbb{L}(y)))$$
$$\mathrm{transformTerm}'(x) \rightarrow ([], x)$$
$$\mathrm{transformTerm}'(y) \rightarrow ([], y)$$
$$\mathrm{transformTerm}'(\mathrm{S}(\mathrm{len}(y))) \twoheadrightarrow ([\mathrm{len}(\mathbb{L}(y), \mathbb{N}(x'))], \mathrm{S}(x'))$$
$$\mathrm{transformTerm}'(\mathrm{len}(y)) \rightarrow ([\mathrm{len}(\mathbb{L}(y), \mathbb{N}(x'))], x')$$

We will discuss the an alternative transformation of the second rule later, when we introduce the constraint solving part of the transformation.

Algorithm 3.3 transformEquation

Require: $(X, f(s_1, \ldots, s_n), g(t_1, \ldots, t_m)) \in \mathcal{E}, f \in M$
Ensure: $term \in \mathfrak{T}$

1 x_1, x_2 are fresh variables;
2 **if** $g \in M$ **then**
3 $(B_f, x_1) := transformTerm'(f(s_1, \ldots, s_n))$;
4 $(B_g, x_2) := transformTerm'(g(t_1, \ldots, t_n))$;
5 $(t'_f, B_f) := pop(B_f)$;
6 $(t'_g, B_g) := pop(B_g)$;
7 $b_f := \square$;
8 **while** $\neg empty(B_f)$ **do**
9 $(b, B_f) := pop(B_f)$;
10 $b_f := b \wedge_p b_f$;
11 **od**
12 $b_g := \square$;
13 **while** $\neg empty(B_g)$ **do**
14 $(b, B_g) := pop(B_g)$;
15 $b_g := b \wedge_p b_g$;
16 **od**
17 $term := t'_f \leftarrow b_f \wedge_p b_g \wedge_p x_1 =_{\mathbb{D}} x_2 \wedge_p t'_g$;
18 **else**
19 $(B_f, x_1) := transformTerm'(f(s_1, \ldots, s_n))$;
20 $(B_g, t'_g) := transformTerm'(g(t_1, \ldots, t_n))$;
21 $(t'_f, B_f) := pop(B_f)$;
22 $b_f := \square$;
23 **while** $\neg empty(B_f)$ **do**
24 $(b, B_f) := pop(B_f)$;
25 $b_f := b \wedge_p b_f$;
26 **od**
27 $b_g := \square$;
28 **while** $\neg empty(B_g)$ **do**
29 $(b, B_g) := pop(B_g)$;
30 $b_g := b \wedge_p b_g$;
31 **od**
32 $term := t'_f \leftarrow b_f \wedge_p b_g \wedge_p x_1 =_{\mathbb{D}} t'_g$;
33 **fi**
34 return(term);

Now we define, that all equations in a specification are transformed to rules in the resulting CLP.

Definition 3.19 (Transformation of Term Rewriting System (TRS) to a CLP). Given $\mathcal{E}$, we define the corresponding CLP (or LP)

$$\mathfrak{P}_{\text{Adt}} := \{\texttt{transformEquation}(\varepsilon) \mid \varepsilon \in \mathcal{E}\}.$$

■

Now, we define two term rewriting relations. The first one is general: Two terms are in a rewrite relation, if the following holds: If a term can be rewritten, its substitution can, too (first rule). And if a term can be rewritten to another term, then this can also happen in the context of a third term (second rule). The second definition defines the same relation for *innermost* term rewriting. Here, a term can only be rewritten, if all of its parameters have already been (substitution rule) or – if it is a parameter itself in the context of a third term – it can be rewritten only if all preceding parameters of the same context have already been rewritten.

Definition 3.20 (Relation $\Rightarrow_{\text{TRS}}$). We write $s \Rightarrow_{\text{TRS}} t$ if there a term s can in one or more steps be rewritten to t. Rewriting is hereby defined as follows:

$$\text{III-a}\ \frac{f(s_1,\ldots,s_n) \to t}{f(s_1^\sigma,\ldots,s_n^\sigma) \Rightarrow_{\text{TRS}} t^\sigma} \qquad \text{III-b}\ \frac{s \Rightarrow_{\text{TRS}} t}{f(\ldots,s,\ldots) \Rightarrow_{\text{TRS}} f(\ldots,t,\ldots)}$$

■

Definition 3.21 (Relation $\Rightarrow_{\text{TRS}_i}$). We write $s \Rightarrow_{\text{TRS}_i} t$ if there a term s can in one or more steps be rewritten to t by an *innermost* rewrite strategy as follows:

$$\text{III-c}\ \frac{f(s_1,\ldots,s_n) \to t \quad s_i^\sigma \not\Rightarrow_{\text{TRS}_i}}{f(s_1^\sigma,\ldots,s_n^\sigma) \Rightarrow_{\text{TRS}_i} t^\sigma} \qquad \text{III-d}\ \frac{s_i \Rightarrow_{\text{TRS}_i} t_i \quad \forall j < i : s_j \not\Rightarrow_{\text{TRS}_i}}{f(\ldots,s_i,\ldots) \Rightarrow_{\text{TRS}_i} f(\ldots,t_i,\ldots)}$$

■

Lemma 3.22. Given a system of equations $\mathcal{E}$ and a corresponding CLP $\mathfrak{P}_{\text{Adt}}$, it holds that

$$s \Rightarrow_{\text{TRS}_i} t \Leftrightarrow \texttt{transformTerm}(s) \Rightarrow_{\text{LP}} \texttt{transformTerm}(t)$$

with $\texttt{transformTerm}(s) \Rightarrow_{\text{LP}} \texttt{transformTerm}(t)$ including substitution steps after the actual resolution of $\texttt{transformTerm}(s)$. ■

The proofs for this lemma can be found in Appendix B.

3.2.2 Semantical Transformation

In the previous section, we defined the syntactical transformation of μCRL ADT equations to pure Prolog rules. However, pure Prolog does not provide full-fledged constraint solving. In order to make use of constraint solving techniques, we have to employ and use Prolog constraint solving extensions. In the remainder of this section, we will define the transformation of a subset of μCRL ADT equations to constraint solving-enabled Prolog rules.

In our approach, constraint solving is used in order to calculate ranges and solutions for numerical or symbolic variables in a system. For constraint solving, we use a CLP(R)-solver, which has the theories for natural numbers, real numbers and enumerative data built in. In the remainder, we will discuss transformation rules for natural numbers as well as for enumerations only.

First of all, we have to consider the conversion of terms of type $\mathfrak{T}_{Nat_p}$ to terms of type $\mathfrak{T}_{Nat_\mu}$. This conversion is achieved by the function transformSuccessor, which transforms a successor term either to an element of the natural numbers, or to a term $c + x$ with $c \in \mathbb{N}$ being a constant and $x \in \mathbb{N}$ being a variable.

Definition 3.23 (Transforming successors). The function

$$\text{transformSuccessor} : \mathbb{N} \times \mathfrak{T}_{Nat_\mu} \rightarrow \mathfrak{T}_{Nat_p}$$

is defined as follows:

$$\text{transformSuccessor}(i, x) = \begin{cases} i & \iff x = 0 \\ i +_p x & \iff x \in X \\ \text{transformSuccessor}(i+1, n) & \iff x = S(n) \end{cases}$$

■

Example 3.24.

1. $\text{transformSuccessor}(0, S(S(0))) \rightarrow \text{transformSuccessor}(2, 0) \rightarrow 2$
2. $\text{transformSuccessor}(0, S(S(x))) \rightarrow \text{transformSuccessor}(2, x) \rightarrow 2 + x$

Assumption 3.25. Successor terms transformed by the function transformSuccessor allow only other successor terms or variables as parameters. Any other type of constructor or function term is not allowed.

This assumption does not affect the generality of the approach, since terms of the form $S(f(x))$ with $f \in M$ and x a variable can in theory be expressed as a term $\text{add}(f(x), S(0))$. As a second step, we extend the algorithm transformTerm′ from the previous section for the treatment of natural numbers. Therefore, we add a case for natural number term to be given as a parameter to the function, leaving the rest unchanged. The according helper function

$$\text{transformTerm}' : \mathfrak{T}(\mathcal{F}, X) \rightarrow \text{Stack}[\mathfrak{T}(\mathcal{F} \cup \mathcal{S} \cup \{=_{\mathbb{D}}\}, X \cup X')] \times \mathfrak{T}(\mathcal{F} \cup \mathcal{S}, X \cup X')$$

is defined in Algorithm 3.4.

Example 3.26. Let $c \in C$ be a constructor. Then:

$$\text{transformTerm}(c(S(S(0)))) \rightarrow x_1 =_{\mathbb{N}} 2 \wedge_p c(\mathbb{N}(x_1))$$
$$\text{transformTerm}'(c(S(S(0)))) \rightarrow ([x_1 =_{\mathbb{D}} 2], c(\mathbb{N}(x_1)))$$
$$\text{transformTerm}(c(S(S(x)))) \rightarrow x_1 =_{\mathbb{N}} 2 + x \wedge_p c(\mathbb{N}(x_1))$$
$$\text{transformTerm}'(c(S(S(x)))) \rightarrow ([x_1 =_{\mathbb{D}} 2 + x], c(\mathbb{N}(x_1)))$$

Algorithm 3.4 transformTerm′

Require: $f(t_1, \ldots, t_n)$
Ensure: $pair \in Stack \times \mathfrak{T}$

```
1  L′ := [];
2  if f :→ ℕ ∧ (f = 0 ∨ f = S) then
3     x is fresh variable;
4     t := transformSuccessor(0, f(t1));
5     pair := (push(x =ℕ t′, []), t′);
6  else if f ∈ M then
7     x is fresh variable;
8     for all 1 ⩽ i ⩽ n do
9        (Li, ti′) := transformTerm′(ti);
10       L′ := pushS(Li, L′);
11    od
12    L := push(f(𝔻1(t1′), ..., 𝔻n(tn′), 𝔻(x)), L′);
13    pair := (L, x);
14 else if f ∈ C then
15    for all 1 ⩽ i ⩽ n do
16       (Li, ti′) := transformTerm′(ti);
17       L′ := pushS(Li, L′);
18    od
19    pair := (L, f(𝔻1(t1′), ..., 𝔻n(tn′), 𝔻(x)));
20 fi
21 return(term);
```

Example 3.27. Now let us come back to the transformation of equation (3.2):

$$\mathrm{len}(\mathrm{list}(x, y)) = S(\mathrm{len}(y))$$

It happens according to the case $g \in M$ from the same definition. Prior to transformation, we have to rewrite the equation as follows:

$$\mathrm{len}(\mathrm{list}(x, y)) = \mathrm{add}(\mathrm{len}(y), S(0)) \tag{3.3}$$

This step is necessary to comply with our assumption about successor terms (Assumption 3.25). Now we can transform the equation:

$$\begin{aligned} \mathrm{transformEquation}(\{\{x, y\}, \mathrm{len}(\mathrm{list}(x, y)), \mathrm{add}(\mathrm{len}(y), S(0))\}) \leftarrow \\ \mathrm{len}(\mathbb{L}(\mathrm{list}(\mathbb{D}_x(x), \mathbb{L}(y))), \mathbb{N}(x_1)) \leftarrow \\ \mathrm{len}(\mathbb{L}(y), \mathbb{N}(x')) \wedge_p x'' =_{\mathbb{N}} 1 \wedge_p \\ x_1 =_{\mathbb{N}} x_2 \wedge_p \mathrm{add}(\mathbb{N}(x'), \mathbb{N}(x''), \mathbb{N}(x_2)) \end{aligned}$$

$$transformTerm'(len(list(x,y))) \twoheadrightarrow ([len(\mathbb{L}(list(\mathbb{D}_x(x),\mathbb{L}(y))),\mathbb{N}(x_1))],x_1)$$
$$transformTerm'(list(x,y)) \twoheadrightarrow ([],list(\mathbb{D}_x(x),\mathbb{L}(y)))$$
$$transformTerm'(x) \rightarrow ([],x)$$
$$transformTerm'(y) \rightarrow ([],y)$$
$$transformTerm'(add(len(y),S(0))) \twoheadrightarrow$$
$$([x'' =_{\mathbb{N}} 1,add(\mathbb{N}(x'),\mathbb{N}(x''),\mathbb{N}(x_2)),len(\mathbb{L}(y),\mathbb{N}(x'))],x_2)$$
$$transformTerm'(len(y)) \rightarrow ([len(\mathbb{L}(y),\mathbb{N}(x'))],x')$$
$$transformTerm'(S(0)) \rightarrow ([x'' =_{\mathbb{N}} 1],x'')$$

Arithmetics

After having defined the basics of term transformation for natural numbers, we will define the semantic transformation of a few functions for this domain. We start with a set of rules for numerical arithmetics. The operator $=_{\mathbb{N}}$ used previously is translated to the according Prolog operator #=.

$$\text{III-e}\ \frac{add : \mathbb{N}\times\mathbb{N}\rightarrow\mathbb{N}}{add(\mathbb{N}(x),\mathbb{N}(y),\mathbb{N}(r)) \leftarrow r\#= x+y}$$

$$\text{III-f}\ \frac{sub : \mathbb{N}\times\mathbb{N}\rightarrow\mathbb{N}}{sub(\mathbb{N}(x),\mathbb{N}(y),\mathbb{N}(r)) \leftarrow r\#= x-y}$$

$$\text{III-g}\ \frac{mult : \mathbb{N}\times\mathbb{N}\rightarrow\mathbb{N}}{mult(\mathbb{N}(x),\mathbb{N}(y),\mathbb{N}(r)) \leftarrow r\#= x*y}$$

$$\text{III-h}\ \frac{div : \mathbb{N}\times\mathbb{N}\rightarrow\mathbb{N}}{div(\mathbb{N}(x),\mathbb{N}(y),\mathbb{N}(r)) \leftarrow r\#= x/y}$$

Arithmetic Comparisons

Here, we define the Prolog rules for comparison of numerical values again for both natural and potentially non-natural numbers. We will begin with the rules defined for positive comparisons (brackets around the bodies of the rules appear here only to improve readability):

$$\text{III-i}\ \frac{eq : \mathbb{N}\times\mathbb{N}\rightarrow\mathbb{B}}{eq(\mathbb{N}(x),\mathbb{N}(y),\mathbb{B}(r)) \leftarrow (x\#= y \rightarrow r=\top;r=\bot)}$$

$$\text{III-j}\ \frac{lt : \mathbb{N}\times\mathbb{N}\rightarrow\mathbb{B}}{lt(\mathbb{N}(x),\mathbb{N}(y),\mathbb{B}(r)) \leftarrow (x\#< y \rightarrow r=\top;r=\bot)}$$

$$\text{III-k}\ \frac{le : \mathbb{N}\times\mathbb{N}\rightarrow\mathbb{B}}{le(\mathbb{N}(x),\mathbb{N}(y),\mathbb{B}(r)) \leftarrow (x\#=< y \rightarrow r=\top;r=\bot)}$$

$$\text{III-l}\ \frac{ge : \mathbb{N}\times\mathbb{N}\rightarrow\mathbb{B}}{ge(\mathbb{N}(x),\mathbb{N}(y),\mathbb{B}(r)) \leftarrow (x\#>= y \rightarrow r=\top;r=\bot)}$$

$$\text{III-m}\ \frac{gt : \mathbb{N} \times \mathbb{N} \to \mathbb{B}}{gt(\mathbb{N}(x), \mathbb{N}(y), \mathbb{B}(r)) \leftarrow (x\#> y \to r = \top; r = \bot)}$$

A separate issue is the handling of the negation operator in Prolog. ECLiPSe Prolog already defines such an operator, however, it cannot be applied in all cases. As an example, have a look at the constraint $x\#< 5$ assuming x being a numerical variable. This constraint restricts the values of x to those values, which are smaller than 5. Intuitively, $not(x\#< 5)$ should restrict the values of x to those, which are greater than or equal to 5, and should thus be equivalent to the expression $x\#>= 5$. However, the constraint solver will simply fail and not return any results for x.

Let us have a look at the reason: Evaluating the constraint $x\# < 5$, the constraint solver tries to prove, that $\exists x : x < 5$. Evaluating the constraint $not(x\# < 5)$, the solver consequently tries to prove $\neg(\exists x : x < 5) \Leftrightarrow \forall x : x \geqslant 5$. This is not true, so the solver fails. In order to surround this problem and to recover the intuitive meaning of the negation operator as it is also used in μCRL, we define a mirror-inverted Prolog rule for each rule defining arithmetic or symbolic comparisons as well as equality in general. The arithmetic comparison rules from above, are thus completed by the following ones (brackets are again inserted to improve the readability):

$$\text{III-n}\ \frac{not(eq(x,y)) \quad not : \mathbb{B} \to \mathbb{B}, eq : \mathbb{N} \times \mathbb{N} \to \mathbb{B}}{neq(\mathbb{N}(x), \mathbb{N}(y), \mathbb{B}(r)) \leftarrow (x\#\backslash= y \to r = \top; r = \bot)}$$

$$\text{III-o}\ \frac{not(lt(x,y)) \quad not : \mathbb{B} \to \mathbb{B}, lt : \mathbb{N} \times \mathbb{N} \to \mathbb{B}}{nlt(\mathbb{N}(x), \mathbb{N}(y), \mathbb{B}(r)) \leftarrow (x\#>= y \to r = \top; r = \bot)}$$

$$\text{III-p}\ \frac{not(le(x,y)) \quad not : \mathbb{B} \to \mathbb{B}, le : \mathbb{N} \times \mathbb{N} \to \mathbb{B}}{nle(\mathbb{N}(x), \mathbb{N}(y), \mathbb{B}(r)) \leftarrow (x\#> y \to r = \top; r = \bot)}$$

$$\text{III-q}\ \frac{not(ge(x,y)) \quad not : \mathbb{B} \to \mathbb{B}, ge : \mathbb{N} \times \mathbb{N} \to \mathbb{B}}{nge(\mathbb{N}(x), \mathbb{N}(y), \mathbb{B}(r)) \leftarrow (x\#< y \to r = \top; r = \bot)}$$

$$\text{III-r}\ \frac{not(gt(x,y)) \quad not : \mathbb{B} \to \mathbb{B}, gt : \mathbb{N} \times \mathbb{N} \to \mathbb{B}}{ngt(\mathbb{N}(x), \mathbb{N}(y), \mathbb{B}(r)) \leftarrow (x\#=< y \to r = \top; r = \bot)}$$

Symbolic Comparisons

Finally, we will define the transformation of comparison equations for symbolic, or enumerative, data. We will only define the transformation of equality and inequality, since any of the other comparison rules implies an order of elements of the particular datatype. Such an order can, however, not be explicitely defined in μCRL (if at all, the order of constructors can be interpreted). The previously used general operator for equality for data of enumerative domains, $=_{\mathbb{E}}$, is translated to its Prolog counterpart $\&=$. The two transformation rules are (brackets are again inserted to improve the readability):

$$\text{III-s}\ \frac{eq : \mathbb{E} \times \mathbb{E} \to \mathbb{B}}{eq(\mathbb{E}(x), \mathbb{E}(y), \mathbb{B}(r)) \leftarrow (x\&= y \to r = \top; r = \bot)}$$

$$\text{III-t}\ \frac{\mathrm{not}(\mathrm{eq}(x,y)) \qquad \mathrm{not}:\mathbb{B}\to\mathbb{B},\mathrm{eq}:\mathbb{E}\times\mathbb{E}\to\mathbb{B}}{\mathrm{neq}(\mathbb{E}(x),\mathbb{E}(y),\mathbb{B}(r)) \leftarrow (x\&\backslash=y \to r=\top; r=\bot)}$$

3.3 Transformation of Process Behavior to Prolog

A parameterized test case may contain traces introduced by data abstraction. Moreover, information about the relationship of symbolic variables or concrete values they can be substituted with is absent. To sort out spurious traces and to obtain information about valuations for symbolic variables, we employ constraint solving.

We transform the original specification $\mathfrak{S}$ to a CLP $\mathfrak{P}$. This CLP can then be queried. A trace π which is selected from, for instance, a test case, is transformed into a query $q := \mathfrak{O}_\pi(\theta)$. Let the set of symbolic variables in the specification be Var_{symb}. If there is no solution for the query, π is a spurious trace, which can be introduced by data abstraction. Such traces are not considered further, since they actually do not exist. If there is a solution θ in $\mathfrak{P}$ for the query, the trace π can be mapped to the trace of the original system.

We refer to trace π with symbolic variables substituted according to θ as an *instantiated trace*, denoted $\pi(\theta)$. The instantiated trace $\pi(\theta)$ is a trace of the original system $\mathfrak{M}$.

$$\text{III-u}\ \frac{\ell \xrightarrow{g\triangleright !s(e)} \hat{\ell} \in E}{s(\mathrm{state}(\ell,\overline{Var}),\mathrm{state}(\hat{\ell},\overline{Var}),\mathrm{param}(e)) \leftarrow g.}$$

$$\text{III-v}\ \frac{\ell \xrightarrow{g\triangleright ?s(x)} \hat{\ell} \in E}{s(\mathrm{state}(\ell,\overline{Var}),\mathrm{state}(\hat{\ell},\overline{Var}_{[x\mapsto y]}),\mathrm{param}(y)) \leftarrow g.}$$

$$\text{III-w}\ \frac{\ell \xrightarrow{g\triangleright x:=c} \hat{\ell} \in E}{\tau(\mathrm{state}(\ell,\overline{Var}),\mathrm{state}(\hat{\ell},\overline{Var}_{[x\mapsto c]}),\mathrm{param}) \leftarrow g.}$$

Table 3.1: Transformation of specification $\mathfrak{S}$ into a CLP $\mathfrak{P}$

Transformation from the original specification $\mathfrak{S} = (L, A, E, (\ell_{init}, \eta_{init}))$ with a set of names E and $s \in E$, $\ell, \hat{\ell} \in L$, G a set of guards with $g \in G$, $x \in Var$, $e \in Exprs(Var)$ to the CLP $\mathfrak{P}$ is defined by the inference rules given in Table 3.1. These rules map edges of the specification to rules of $\mathfrak{P}$. All the rules are of the form $\rho(\mathrm{state}(\ell,\overline{Var}),\mathrm{state}(\hat{\ell},\overline{Var}'),\mathrm{param}(y)) \leftarrow g$. The first state parameter describes the source state of the edge in terms of the specification location and the process variables. The second state parameter describes the changed target state in the same terms. The third parameter param contains all symbolic variables or expressions which are local for this edge. These are the action parameters.

Rule III-u transforms an output edge $\ell \xrightarrow{g \triangleright !s(e)} \hat{\ell}$ into a rule

$$s(state(\ell, \overline{Var}), state(\hat{\ell}, \overline{Var}), param(e)) \leftarrow g.$$

The name of the rule coincides with the signal s. The edge leads to a change of location from ℓ to $\hat{\ell}$. The values of the process variables $\overline{Var}$ remain unmodified. The signal is parameterized with a value given by expression e that becomes a parameter of the $param$-part of the rule. The rule holds only if the guard is satisfied.

Rule III-v transforms an input edge $\ell \xrightarrow{g \triangleright ?s(x)} \hat{\ell}$ into a rule

$$s(state(\ell, \overline{Var}), state(\hat{\ell}, \overline{Var}_{[x \mapsto y]}), param(y)).$$

Here, input leads to the substitution of process variable x by a symbolic variable y that is local for this rule.

Rule III-w maps an assign-edge $\ell \xrightarrow{g \triangleright x := e} \hat{\ell}$ into a τ-rule

$$\tau(state(\ell, \overline{Var}), state(\hat{\ell}, \overline{Var}_{[x \mapsto e]}), param) \leftarrow g.$$

The rule is satisfied only if the guard g is satisfied. An assignment is represented by substituting process variable x by expression e. τ-rules have no local parameters, thus the structure $param$ has the arity 0.

3.4 Related Work

The use of constraint solvers in symbolic testing or model-checking approaches is in general quite common. This always leads to the question, how certain specifications can be correctly and efficiently be transformed into CLPs for certain constraint solvers. In this section, we will thus not lead a general discussion of this topic, but turn our focus to one particular approach which also uses Prolog-based techniques in order to do constraint solving on formal specifications.

The approach under consideration is the transformation of specifications in AutoFocus to CLPs developed by Lötzbeyer and Pretschner (2000). AutoFocus is a CASE-tool (Computer-Aided Software Engineering tool) based on the specification language Focus for distributed systems. In Focus (Broy, 1998), a system is modeled as a set of components, which are composed by typed channels. The relation of the histories of a component's input and output streams define the behavior of this component in the system.

Technically, the approach by Lötzbeyer and Pretschner transforms AutoFocus specifications into Constraint Handling Rules (CHR), a high-level language which allows the definition of customized constraint solvers and the extension of existing ones. Our approach, defined in this Chapter, is built upon the standard Prolog mechanisms and the *ic* constraint solver for natural numbers. The main differences beyond technicalities of the two approaches can be located in the two areas of data and behavior representation in the CLP.

User-defined datatypes in Focus are provided in a Gofer-like syntax. μCRL makes use of a syntax which is based on equations and explicitly makes use of term rewriting techniques. As a consequence, our approach simulates term rewriting, esp. for structured datatypes and nested function calls. Arithmetic expressions on numerical or symbolic variables and values are translated to the respective Prolog constraint solver operators in both approaches, with Lötzbeyer and Pretschner doing an in place translation of these constraints. In contrast, we keep the original expressions from the specification in place and transparently provide a custom realization of the respective operations as rules in the CLP. This provides us with more flexibility w.r.t. possible manual or automatic adjustments of the operations directly in the CLP. Typing of data elements in the AutoFocus approach is provided by a rule `isType` for each datatype. This, of course, leads to an additional rule invocation for each datatype lookup during constraint solving. Our approach of an in place typing does not require this extra invocation, but allows Prolog to match types on the fly.

The behavior of components in AutoFocus is defined in state transition diagrams. Those are in principle comparable to resp. IOSTSs and μCRL specifications. A significant difference is that transitions in AutoFocus contain labels with explicit input and output statements rather than parameterized action calls. These statements hold a pair of a channel name and a pattern or expression. This difference is due to the fact that system behavior in AutoFocus is defined by the history of channels rather than traces of action calls. This fact also has consequences for the definition of Prolog rules for action steps in the system. The AutoFocus approach defines only one procedure named `step` with all the further information about particular transitions in the system being given as parameters of the respective rules. We, however, directly name the Prolog rules after the respective actions, which allows Prolog to look up the rule needed for a particular transition in the system at least as fast as the AutoFocus approach. The approach for AutoFocus specifications has the advantage of a higher flexibility, probably with the background that parallel composition of components is defined on the level of Prolog. We do not need this flexibility in our approach, since parallel composition of processes happens in the μCRL specification already.

Idle transitions in AutoFocus need to be complemented by special rules; this is not necessary for μCRL. However, also for μCRL complementation of rules is connected to the negation problem of Prolog like it is for the AutoFocus approach. The problem is solved in a quite similar way in both approaches, however, our approach is a bit more general, since it does not only consider (in)equalities, but the negation of all possible boolean expressions, as can also be found later in Section 5.2.4. This seems not to be the case for Lötzbeyer and Pretschner (2000).

The two approaches, we have just compared, are also used in different settings. The AutoFocus approach is used by Pretschner et al. (2004a,b) for the generation of test cases. Our approach is used for test data selection only, while the generation part is undertaken by the enumerative approach based on LTSs, which we will discuss in Chapter 4. An actual trace computation by the constraint solver takes place only in the context of model checking and bug hunting with false negatives, as it will be discussed in Chapter 7.

Chapter 4

Testing with Data Abstraction

> Als de computer mij een uitkomst geeft die zegt dat ik zes meter naast mijn baanvlak zit dan wil ik er drie meter van geloven maar niet alle zes.
>
> *(Buzz Aldrin)*

The test of a software product is a crucial aspect in every software development process. In today's form as an approach to show up failures in a software, it has been established by Myers (1979). There is no single activity named "testing" with one well-defined semantics, however, there is a multidimensional typology of testing as can, for instance, be found in the thesis of Brandán Briones (2007, Section 1.1). Let us first have a brief look at this typology, in order to position our further work in this and the next chapter.

One dimension of the typology regards the different quality characteristics, which we had already mentioned in the beginning of Chapter 1. It defines the *test target*, which can be roughly categorized into functional requirements (suitability, accuracy, interoperability, security and compliance with ISO 9126; cf. Van Veenendaal (2002), Appendix B), and non-functional requirements like the performance of the tested software or its robustness (ibid).

The second dimension regards the visibility of internals of the System under Test (SUT) to the tester. Here, we distinguish whitebox, graybox and blackbox testing. While whitebox tests are based on the interna of the SUT, like its source code, which are completely visible to the tester, blackbox tests are restricted to information that is available externally to the SUT. In many cases, this is its specification and interface definitions. Graybox tests, finally, can refer to more information than just that of the system specification, but still not to all interna of the SUT.

The third dimension is the position of a particular test in the software development process. This becomes especially clear in the V-Model as introduced by Boehm (1979), which explicitly distinguishes the component test, the integration test, the system test and finally the acceptance test. While the component test validates single components of the SUT, these components get gradually integrated during the integration test until the whole system is tested at once (system test). While these three tests are accomplished by the contractor, who produces the software system, the last, the acceptance test, is accomplished together with or even just by the customer of the software.

In this thesis, we concentrate on the blackbox system test, targeting functional requirements. In order to validate functional requirements of a system, we apply

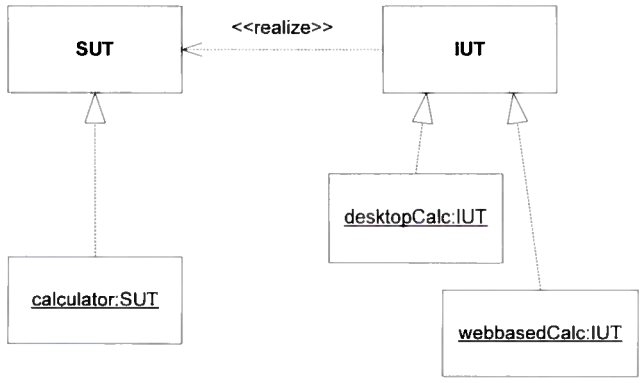

Figure 4.1: Realization relation between SUT and IUT

model-based testing. This approach emanates from a model or a specification of the system, and validates the IUT against this specification. It is therefore a test-last approach, since we do not use the test cases as the specification of the system's functionality.

The Difference between SUTs and IUTs

At this point, the well-disposed reader might have found out, that we switch between the terms System under Test (SUT) and Implementation under Test (IUT) from time to time. In most literature, you will find either SUT or IUT, however, such a decision seemed not to be sufficient in our case.

Figure 4.1 shows the relation between the two terms as we want to understand it in this thesis by means of a small example. The requirements and specifications of the calculator in the figure form the SUT, which will be considered during test generation. This SUT is at some point realized by an IUT. This IUT can have different characteristics, as in this example shown by a desktop-based and a web-based implementation of the SUT. Test execution is then performed against one of these implementations.

Conformance Testing Conformance testing (ITU-T, 1996) is one of the most rigorous among existing testing techniques, checking whether an IUT is consistent with its functional specification. Roughly speaking, this is only the case if every observable behavior of the IUT is allowed by the specification. Test cases are generated from the system's specifications. In some cases, this process is guided by so-called *test purposes*, which are sketches of possible test scenarios. The generated test cases do not reject consistent IUTs, and do not accept IUTs showing behaviors not allowed by the specification. Two major problems of automatic test generation are its termination and the number of generated test cases. If the generation process is not guided except for the specification documents, it may not terminate at all or produce many

unnecessary test cases around a few useful ones. Test purposes allow us to focus the generation on certain aspects, like the main risks of a system (*risk-based testing*).

Not only the selection of behavior is crucial for successful test generation, but also that of data. Software in most cases interacts with an environment, which stimulates the execution of interface actions parameterized with data values coming from large or even infinite domains. Considering these parameters already at the stage of test generation leads to problems for enumerative techniques concerning the state space of the system – state space explosion was already mentioned in the introduction, Chapter 1. It also reduces the reusability of the resulting test cases. For this reason, we have to abstract away from concrete data and concentrate on behavioral aspects only for test case generation, and reintroduce data for test execution only.

In this chapter, we present a test generation framework (Calamé et al., 2005, 2007a). Starting from the specification of an SUT and an appropriate test purpose, we abstract away input and output data from the specification using the idea of *chaotic data abstraction* by Sidorova and Steffen (2001a). The abstract system then shows at least the behavior of the original system, as has been worked out by Ioustinova et al. (2002b). Afterwards, *abstract* test cases are generated, which contain a control flow and are parameterizable with concrete data values during test execution. These data values can be obtained from a CLP, which is set up in parallel and produces data intervals for data selection and serves as a test oracle.

We implement our approach to generate test cases from μCRL specifications. Chaotic data abstraction has first been proposed by Sidorova and Steffen for specifications in SDL (Specification and Description Language). In this chapter, we develop a chaotic data abstraction for μCRL. In order to do so, we define abstracted datatypes besides the original ones, and lifting functions from the original datatypes to their respective abstracted counterparts. The necessary propagation of abstracted data values can happen on the level of summands in μCRL. In order to overapproximate the original systems by the abstracted one, guards for transitions must be realized with a *may*-semantics. This semantics has originally been introduced by Larsen and Thomsen (1988), has then been implemented for SDL by Ioustinova et al. (2002a, 2004) and is in this chapter realized for the process algebra μCRL.

The actual generation of test cases is performed by the tool TGV. TGV (Jard and Jéron, 2005) is an automatic generator of test cases from formal specifications of reactive systems. TGV implements algorithms based on adaptation of on-the-fly model-checking algorithms. Test selection in TGV is based on the concept of test purposes. However, specifications of systems operating on large or infinite data domains are beyond the scope of TGV, even with on-the-fly test generation using test purposes.

This chapter is organized as follows. In Section 4.1, we give a formal introduction to conformance testing, which our approach is based on. The test generator TGV is introduced in Section 4.2. Then, the approach of data abstraction and test generation with TGV is worked out in Section 4.3. Section 4.4 discusses the determination of test case parameters using constraint solving. In Section 4.5, we work out the application of our approach to the CEPS as defined by the CEPSCO (2000). Finally, in Section 4.6, we will discuss related work in the field of model-based testing.

4.1 Conformance Testing Theory: ioco

Our approach is based on conformance testing that validates whether an implementation conforms to its specification. In a theory of conformance testing by Tretmans (1996), the notion of conformance is formalized by a *conformance relation* between specification and implementation that are assumed to be IOLTSs. Tretmans describes several such implementation and conformance relations, of which the most rigorous one is the relation ioco.

In principle, conformance means that – given specified input – an implementation produces at most the output, which had been specified in its specifications. We will first define a function, which provides all possible system output for a given set of states and then define a relation of conformance for a specification $\mathfrak{M}$ and an implementation $\mathfrak{I}$.

Definition 4.1 (Function out; Tretmans, 1996). Let $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{\text{init}})$ be an IOLTS. The function $\text{out} : 2^{\Sigma} \to 2^{\Lambda}$ is defined for a set of states $\Sigma' \subseteq \Sigma$ as: $\text{out}(\Sigma') = \{\lambda | \lambda \in \Lambda_{\text{out}} \wedge \exists \sigma \in \Sigma'. \exists \hat{\sigma} \in \Sigma : \sigma \xrightarrow{\lambda} \hat{\sigma}\}$. ■

Definition 4.2 (ioconf; ibid). Let $\mathfrak{M}$ be an IOLTS and let $\mathfrak{I}$ be its input-complete implementation. Then

$$\mathfrak{I} \text{ ioconf } \mathfrak{M} \Leftrightarrow \forall \pi \in [\![\mathfrak{M}]\!]_{\text{traces}} : \text{out}(\mathfrak{I} \text{ after } \pi) \subseteq \text{out}(\mathfrak{M} \text{ after } \pi)$$

■

This relation covers already quite a lot of cases, however, it does not properly consider the *one* case: that the implementation $\mathfrak{I}$ does *not* produce output after a particular input. According to the above definition, an implementation $\mathfrak{I}$ conforms to its specification $\mathfrak{S}$, $\mathfrak{I}$ ioconf $\mathfrak{M}$, even if it does not produce *any* output: $\forall \pi \in [\![\mathfrak{M}]\!]_{\text{traces}} : \emptyset \subseteq \text{out}(\mathfrak{M} \text{ after } \pi)$. While for the licensee of a gambling machine such a behavior might realize great income, a bank operating ATMs which behave like that would for sure lose customers. To make a long story short: We need a way to express the absence of output and to include it in the conformance relation.

The solution is a stronger variant of ioconf, ioco, which does exactly this. First of all, it introduces the notion of *quiescence* in order to describe absent output, then it extends ioconf by the handling of quiescence. Quiescence itself is originally defined as follows:

Definition 4.3 (Quiescence; Tretmans, 1996; Vaandrager, 1991). Let $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{\text{init}})$ be an IOLTS with $\Lambda = \Lambda_{\text{in}} \cup \Lambda_{\text{out}} \cup \Lambda_{\tau}$. Quiescence is defined as a function $\text{quiescent} : \Sigma \to \mathbb{B}$. For a state $\sigma \in \Sigma$ holds:

$$\text{quiescent}(\sigma) = \neg \exists \lambda \in \Lambda_{\text{out}} \cup \Lambda_{\tau} : \sigma \xrightarrow{\lambda} \hat{\sigma} \in \Delta \text{ with } \hat{\sigma} \in \Sigma$$

■

In Tretmans (1996), the action $\delta \notin \Lambda$ is introduced to denote a deadlock for $\sigma \in \Sigma$ with $\text{quiescent}(\sigma)$. Traces ending in quiescence are named *quiescent traces*.

Definition 4.4 (Quiescent Trace; Tretmans, 1996). A quiescent trace is a trace $\pi.\delta \in [\![\mathfrak{M}]\!]_{traces}$, i.e. a trace ending in a quiescent state. ■

For the moment, a state is quiescent, if it has only internal outgoing transitions (τ-steps), and a trace is quiescent, if it ends in a quiescent state. This is inconvenient, since we want to use quiescence to denote the absence of *output* rather than that of *external* transitions. Furthermore, we are not interested to let a trace end at the first occurrence of quiescence, but want to be able to provide input and follow the trace further. So on the way to a definition of ioco, there lie still the definitions of traces which allow the occurrence of quiescence not only at their end, but also midway: failure traces and suspension traces.

Definition 4.5 (Failure Trace; ibid). A failure trace contains both actions as transitions $\sigma \xrightarrow{\lambda} \hat{\sigma}$, and refused actions λ_{ref} as self loops $\sigma \xrightarrow{\lambda_{ref}} \sigma$. The set of failure traces for $\mathfrak{M}$ is $[\![\mathfrak{M}]\!]_{Ftraces} = \{\pi \in (\Lambda \cup 2^{\Lambda})^* | \mathfrak{M} \xrightarrow{\pi}\}$. ■

Definition 4.6 (Suspension Trace; ibid). A suspension trace is a trace $\pi \in [\![\mathfrak{M}]\!]_{Ftraces} \cap (\Lambda \cup \{\delta\})^*$. The set of suspension traces is $[\![\mathfrak{M}]\!]_{Straces}$. ■

While a quiescent trace thus only ends in quiescence (in a deadlock), a suspension trace can contain δ-loops repetitively. The occurrence of a δ-loop means, that input must be given to the machine. The action δ is handled as explicit, observable output.

With these definitions, we can extend Definition 4.2 by the notion of absent output from the implementation $\mathfrak{I}$:

Definition 4.7 (ioco; ibid).

$$\mathfrak{I} \text{ ioco } \mathfrak{M} \Leftrightarrow \forall \pi \in [\![\mathfrak{M}]\!]_{Straces} : \text{out}(\mathfrak{I} \text{ after } \pi) \subseteq \text{out}(\mathfrak{M} \text{ after } \pi)$$

■

In the remainder of this chapter, we will not be concerned by a full ioco. The test generator TGV can explicate the above-mentioned "output" of absent output for systems specified as LTSs by introducing loops $\sigma \xrightarrow{\text{OUTPUTLOCK}} \sigma$ for a state σ. However, for systems with data, Zinovieva-Leroux (2004) argues that quiescence is in principal undecidable, because it can be induced by τ-loops in a system (livelocks), which might not be detectable in finite time. For this reason, Zinovieva-Leroux decides to regard the theory ioc instead, which is practically ioconf lifted to systems with data. A theory comparable to ioco can, according to Zinovieva-Leroux, only be strived for by prohibiting livelocks in the syntax of the system specification. However, even though this allows to generate a suspension automaton for the system – as TGV does to explicate absent output –, a full blocking detection on the level of IOSTS is still untrackable.

As a practical solution for this problem, Zinovieva-Leroux proposes to use a timer during test execution. If this timer times out, then the tester can decide to no longer wait for any output from the IUT. However, this decision can, of course, easily be a wrong one, if the timer duration is chosen too short. We will come back to this topic

in Chapter 5, where we work out the execution of tests. There, we will discuss the treatment of absent output during test execution with a outlook on timed versions of ioco.

4.2 Test Generation with TGV

IOLTSs modeling IUTs are assumed to be *input complete*, meaning, the implementation cannot refuse any input from the environment. Given a model $\mathfrak{I}$ of an implementation and a model $\mathfrak{M}$ of a specification, the implementation *conforms* to the specification if and only if for each trace π in $[\![\mathfrak{M}]\!]_{traces}$, $\mathfrak{I}$ after π produces only outputs that can be produced by $\mathfrak{M}$ after π. In case, $\mathfrak{M}$ is input complete, conformance is the standard trace inclusion relation.

We are interested in test generation where the test selection is guided by a test purpose (Jard and Jéron, 2005). In order to define the notion of test purposes, we have to define that of *trap states*. A trap state is a state, which cannot be left anymore by any of its outgoing transitions.

Definition 4.8 (Trap State). In an IOLTS $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{init})$, a *trap state* is a state $\sigma \in \Sigma$ for which $trap : \Sigma \to \mathbb{B}$ defined as $trap(\sigma) = \forall \sigma \xrightarrow{\lambda} \sigma' \in \Delta : \sigma = \sigma'$, holds. ■

A test purpose forms a sketch of the scenarios for the test cases, which TGV is supposed to generate. It is a deterministic IOLTS $\mathfrak{M}_{TP}$ that is equipped with a non-empty set of accepting states Σ_{acc} and a set of refusing states Σ_{ref} which can be empty. These two sets form the end points of traces, which are allowed or explicitly unwanted in the test, resp. Both accepting and refusing states are trap states. Moreover, $\mathfrak{M}_{TP}$ is input complete in all the states except of the accepting and refusing ones.

Definition 4.9 (Test Purpose). Let $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{init})$ be a specification. A test purpose is a *deterministic* IOLTS $\mathfrak{M}_{TP} = (\Sigma^{TP}, \Lambda^{TP}, \Delta^{TP}, \sigma^{TP}_{init})$ with a set of labels $\Lambda^{TP} = \Lambda_I \cup \Lambda_O \cup \{\lambda_{acc}, \lambda_{ref}\}$. Internal actions of $\mathfrak{S}$ are not considered here.

Let furthermore be:

$$\begin{aligned} \Sigma^{TP}_{acc} &= \{\sigma \in \Sigma^{TP} | trap(\sigma) \wedge \exists t \in \Delta^{TP} : t = (\sigma, \lambda_{acc}, \sigma)\} \text{ and} \\ \Sigma^{TP}_{ref} &= \{\sigma \in \Sigma^{TP} | trap(\sigma) \wedge \exists t \in \Delta^{TP} : t = (\sigma, \lambda_{ref}, \sigma)\}. \end{aligned}$$

Σ^{TP}_{acc} is the set of *accepting states* of the test purpose, Σ^{TP}_{ref} the set of *refusing states*. The following must hold for a test purpose:

$$\Sigma^{TP}_{acc} \neq \emptyset \wedge \Sigma^{TP}_{acc} \cap \Sigma^{TP}_{ref} = \emptyset.$$

Transitions labeled with λ_{acc} or λ_{ref} are allowed in trap states only. ■

A test purpose does not have to be designed as a complete test purpose by the test engineer. A complete test purpose has outgoing transitions for *all* action labels from the original specification in each of its states. A test purpose can rather be designed incompletely, i.e. each state in the test purpose has only outgoing transitions for a

subset of the set of action labels of the original specification. In this case, the test purpose is completed by TGV prior to test generation. Therefore, TGV introduces a $*$-loop $\sigma \xrightarrow{*} \sigma$ in each state σ of the test purpose. This $*$-loop is further expanded in a way, that a complete test purpose is retrieved. This complete test purpose is then used in the further test generation process.

Definition 4.10 (Semantics of $*$-loops). Let $\mathfrak{M}_{TP} = (\Sigma^{TP}, \Lambda^{TP}, \Delta^{TP}, \sigma_{init}^{TP})$ be a test purpose for $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{init})$. Let furthermore $\mathfrak{M}_{TP'} = (\Sigma^{TP}, \Lambda^{TP} \cup \{*\}, \Delta^{TP'}, \sigma_{init}^{TP})$ be this test purpose, with $\Delta^{TP'} = \Delta^{TP} \cup \{\sigma \xrightarrow{*} \sigma | \sigma \in \Sigma^{TP}\}$ the $*$-loops implicitely introduced by TGV.

Let furthermore $\hat{\sigma}, \hat{\hat{\sigma}} \in \Sigma^{TP}$ be two states in this test purpose. The semantics of a $*$-loop $\hat{\sigma} \xrightarrow{*} \hat{\sigma}$ is defined by the function $expand : \Delta^{TP'} \to 2^{\Sigma^{TP} \times \Delta^{TP} \times \Sigma^{TP}}$ as follows:

$$expand(\hat{\sigma} \xrightarrow{\lambda} \hat{\hat{\sigma}}) = \{\hat{\sigma} \xrightarrow{\lambda} \hat{\hat{\sigma}}\} \cup \{\hat{\sigma} \xrightarrow{\lambda'} \hat{\sigma} | \lambda' \in \Lambda \wedge \neg \exists \hat{\sigma} \xrightarrow{\lambda'} \hat{\hat{\sigma}} \in \Delta^{TP}\}$$

■

Test generation guided by a test purpose consists in building the synchronous product of the system and the test purpose $\mathfrak{M}_{SP} = \mathfrak{M} \times \mathfrak{M}_{TP}$, and finally transforming it into a Complete Test Graph (CTG) $\mathfrak{M}_{CTG}$ by assigning verdicts. The state space of the synchronous product $\mathfrak{M}_{SP}$ forms the reachable part of $\Sigma \times \Sigma^{TP}$. The set Δ^{SP} is constructed by matching the labels of transitions in $\mathfrak{M}$ and $\mathfrak{M}^{TP}$. By doing so, the complete behavior of the specification $\mathfrak{M}$ is reduced to scenarios sketched by the test purpose.

Definition 4.11 (Synchronous Product of $\mathfrak{M}$ and $\mathfrak{M}_{TP}$; Jard and Jéron, 2005). The synchronous product $\mathfrak{M}_{SP}$ of the system specification $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{init})$ and a input complete test purpose $\mathfrak{M}_{TP} = (\Sigma^{TP}, \Lambda^{TP}, \Delta^{TP}, \sigma_{init}^{TP})$ is the IOLTS $\mathfrak{M}_{SP} = \mathfrak{M} \times \mathfrak{M}_{TP} = (\Sigma^{SP}, \Lambda^{SP}, \Delta^{SP}, \sigma_{init}^{SP})$ where:

- $\Lambda^{SP} = \Lambda \cup \{\lambda_{acc}, \lambda_{ref}\}$ is the alphabet of the IOLTS,
- Δ^{SP} is the set of transitions such that

$$(\sigma, \sigma'') \xrightarrow{\lambda} (\sigma', \sigma''') \in \Delta^{SP} \Leftrightarrow \Big((\sigma \xrightarrow{\lambda} \sigma' \in \Delta \wedge \sigma'' \xrightarrow{\lambda} \sigma''' \in \Delta^{TP}) \vee (\sigma'' \xrightarrow{\lambda_{acc}} \sigma''' \in \Delta^{TP} \wedge \sigma'' = \sigma''') \vee (\sigma'' \xrightarrow{\lambda_{ref}} \sigma''' \in \Delta^{TP} \wedge \sigma'' = \sigma''') \Big),$$

  and
- $\sigma_{init}^{SP} = (\sigma_{init}, \sigma_{init}^{TP}) \in \Sigma^{SP}$ is the initial state.

■

For the moment, the synchronous product contains the system behavior and some additional information, which traces are acceptable and which should be refused. However, in order to produce test cases, in principle two more things have to happen:

1. The focus has to be changed from the SUT to an external *tester*, i.e. inputs and outputs have to be mirrored.
2. The additional information on accepted and refused traces must be translated in actual *test verdicts*.

TGV takes these steps in order to produce a CTG, which can already be used for testing the system.

Definition 4.12 (Complete Test Graph; Jard and Jéron, 2005). The CTG is an IOLTS $\mathfrak{M}_{CTG} = (\Sigma^{CTG}, \Lambda^{CTG}, \Delta^{CTG}, \sigma^{CTG}_{init})$, which is determined from the synchronous product $\mathfrak{M}_{SP}$ in the following way:

1. The set of actions is determined by mirroring the set of actions of $\mathfrak{M}_{CTG}$: $\Lambda^{CTG} = \Lambda^{CTG}_I \cup \Lambda^{CTG}_O$ with $\Lambda^{CTG}_O \subseteq \Lambda_I$ and $\Lambda^{CTG}_I = \Lambda_O$.
2. The set of states is determined. This set is divided into four subsets $\Sigma^{CTG} = \Sigma^{CTG}_{L2A} \dot\cup \Sigma^{CTG}_{Inconc} \dot\cup \Sigma^{CTG}_{Fail}$, which are defined as follows:

   Lead to Accept: $\Sigma^{CTG}_{L2A} = \{\sigma \in \Sigma^{SP} | \exists \pi \in [\![\mathfrak{M}_{SP}]\!]_{traces} (\sigma \xrightarrow{\pi} \sigma' \wedge \sigma' \in \Sigma^{SP}_{acc})\}$,

   Pass: The set $\Sigma^{CTG}_{Pass} \subseteq \Sigma^{CTG}_{L2A}$ is defined as $\Sigma^{CTG}_{Pass} = \Sigma^{SP}_{acc}$. This set must be non-empty.

   Inconclusive: $\Sigma^{CTG}_{Inconc} = \{\sigma' | \exists \sigma \in \Sigma^{CTG}_{L2A}, \sigma' \notin \Sigma^{CTG}_{L2A}, \iota \in \Lambda^{SP}_O (\sigma \xrightarrow{\lambda} \sigma' \in \Delta^{SP})\}$,

   Fail: $\Sigma^{CTG}_{Fail} = \{\sigma^{CTG}_{Fail}\}$, $\sigma^{CTG}_{Fail} \notin \Sigma^{SP}$.

   For reasons of manageability of the resulting IOLTS, the state σ^{CTG}_{Fail} exists only implicitly and is assumed as end point for all possible traces $\sigma \notin [\![\mathfrak{M}_{SP}]\!]_{traces}$. It is not actually generated.
3. The set of transitions of the CTG is defined as $\Delta^{CTG} = \Delta^{CTG}_{L2A} \cup \Delta^{CTG}_{Inconc} \cup \Delta^{CTG}_{Fail}$ with:

$$\begin{aligned}
\Delta^{CTG}_{L2A} &= \Delta^{SP} \cap (\Sigma^{CTG}_{L2A} \times \Lambda^{CTG} \times \Sigma^{CTG}_{L2A}),\\
\Delta^{CTG}_{Inconc} &= \Delta^{SP} \cap (\Sigma^{CTG}_{L2A} \times \Lambda^{CTG}_I \times \Sigma^{CTG}_{Inconc}),\\
\Delta^{CTG}_{Fail} &= \{\sigma \xrightarrow{\lambda} \sigma^{CTG}_{Fail} | \sigma \in \Sigma^{CTG}_{L2A} \wedge \lambda \in \Lambda^{CTG}_I \wedge \sigma \text{ after } \lambda = \emptyset\}.
\end{aligned}$$

■

A CTG may contain loops and choices between several outputs in the same state or between inputs and outputs, and is thus *not (necessarily) controllable*.

During generation of the $\mathfrak{M}_{CTG}$, all input and output actions are mirrored, so that the set of input actions of the $\mathfrak{M}_{CTG}$ equals the set of output actions of the $\mathfrak{M}$ and the set of output actions of the $\mathfrak{M}_{CTG}$ is a subset of the set of input actions of the $\mathfrak{M}$ (Figure 4.2). The reason for mirroring inputs and outputs lies in the relation between a test case and the IUT, as the input of the IUT is the output of the test case and vice versa. However, since a test case can normally not test all possible inputs of an SUT, its set of outputs Λ^{CTG}_O is limited to a subset of the SUT's set of inputs Λ_I.

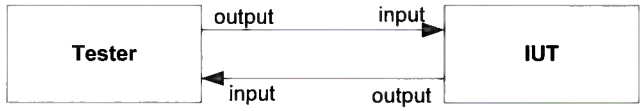

Figure 4.2: Mirroring inputs and outputs on the tester

The sets of accepting and refusing states of $\mathfrak{M}_{SP}$ induce the sets of accepted and refused traces, denoted $[\![\mathfrak{M}_{SP}]\!]_{atrace}$ or $[\![\mathfrak{M}_{SP}]\!]_{rtrace}$, resp., where $[\![\mathfrak{M}_{SP}]\!]_{atrace} \subseteq [\![\mathfrak{M}]\!]_{traces}$ and $[\![\mathfrak{M}_{SP}]\!]_{rtrace} = [\![\mathfrak{M}]\!]_{traces} \setminus [\![\mathfrak{M}_{SP}]\!]_{atrace}$. Depending on the trace, executed during the actual test, a verdict is assigned.

Definition 4.13 (Verdict). A *verdict* is the result of the execution of a test case. It is determined by the comparison between the actual behavior of the IUT during test case execution and its expected behavior. In general, there exist five types of verdicts of which we consider the following four: Pass, Inconc, Fail and None.

The verdict is set by a function $\mathit{setverdict} : [\![\mathfrak{M}_{SP}]\!]_{traces} \to \mathit{Verdict}$, which is defined as follows:

$$\mathit{setverdict}(\pi) = \begin{cases} \mathit{Pass} & \iff \pi \in [\![\mathfrak{M}_{SP}]\!]_{atrace} \\ \mathit{Inconc} & \iff \pi \in [\![\mathfrak{M}_{SP}]\!]_{rtrace} \\ \mathit{Fail} & \iff \pi \notin ([\![\mathfrak{M}_{SP}]\!]_{atrace} \cup [\![\mathfrak{M}_{SP}]\!]_{rtrace}) \wedge |\pi| > 0 \\ \mathit{None} & \iff |\pi| = 0 \end{cases}$$

The types of verdicts can be structured in the following partial order:

$$\mathit{None} \sqsubseteq \mathit{Pass} \sqsubseteq \mathit{Inconc} \sqsubseteq \mathit{Fail}$$

■

The fifth type of verdict, Error, is not considered here: It is assigned only, if the test run itself runs into an undefined (exceptional) state. In the above partial order relation, it resides above Fail: $\mathit{Fail} \sqsubseteq \mathit{Error}$.

The Pass verdict is assigned to those states of $\mathfrak{M}_{CTG}$, which correspond to the final states of traces from $[\![\mathfrak{M}_{SP}]\!]_{atrace}$ and thus to the *accepting* states in the test purpose. The Inconc verdict is assigned to states from which accepting states are not reachable. In this case, the state is still on a trace of $\mathfrak{M}$, but the trace does not satisfy the test purpose (traces from $[\![\mathfrak{M}_{SP}]\!]_{rtrace}$). The Fail verdict is implicit. All unspecified output leads to this verdict.

When is a test successful?

The answer to this question has changed over the years with the changing objective of testing, as discussed in the introduction (Chapter 1). While in earlier years a test might have been successful when the functionality of the IUT could be confirmed, e.g. by a number of tests resulting in a Pass verdict, this has changed significantly. Regarding the fact that a test serves as a fault detector (Myers, 1979), it is only successful if it leads to a Fail verdict.

Since this fact might be irritating for the reader, and since it is not really obvious which verdict is then assigned to a *failing* test, we will use neither of these expressions in this thesis. Rather, we will name the verdict to which a particular test leads.

As we said before, $\mathfrak{M}_{CTG}$ may contain choices between several outputs and choices between inputs and outputs. Controllable test cases are derived by resolving these choices, meaning, a test case does not contain these choices between outputs or between inputs and outputs anymore.

Definition 4.14 ((Controllable) Test Case). A test case is a deterministic input complete IOLTS $\mathfrak{M}_{TC} = (\Sigma^{TC}, \Lambda^{TC}, \Delta^{TC}, \sigma^{TC}_{init})$ derived from $\mathfrak{M}_{CTG}$ with

$$\begin{aligned}
&\Sigma^{TC} \subseteq \Sigma^{CTG} : \Sigma^{TC} = \Sigma^{TC}_{L2A} \cup \Sigma^{TC}_{Pass} \cup \Sigma^{TC}_{Inconc} \cup \Sigma^{TC}_{Fail} \text{ and} \\
&\qquad\qquad \Sigma^{TC}_{L2A} \subseteq \Sigma^{CTG}_{L2A}, \Sigma^{TC}_{Pass} \subseteq \Sigma^{CTG}_{Pass}, \Sigma^{TC}_{Inconc} \subseteq \Sigma^{CTG}_{Inconc}, \Sigma^{TC}_{Fail} \subseteq \Sigma^{CTG}_{Fail}; \\
&\Lambda^{TC} \subseteq \Lambda^{CTG} : \Lambda^{TC}_{O} \subseteq \Lambda^{CTG}_{O} \wedge \Lambda^{TC}_{I} = \Lambda^{CTG}_{I}; \\
&\Delta^{TC} \subseteq \Delta^{CTG} : \Delta^{TC} = \Delta^{TC}_{L2A} \cup \Delta^{TC}_{Inconc} \cup \Delta^{TC}_{Fail} \text{ and} \\
&\qquad\qquad \Delta^{TC}_{L2A} \subseteq \Delta^{CTG}_{L2A}, \Delta^{TC}_{Inconc} \subseteq \Delta^{CTG}_{Inconc}, \Delta^{TC}_{Fail} \subseteq \Delta^{CTG}_{Fail}; \\
&\sigma^{TC}_{init} = \sigma^{CTG}_{init}.
\end{aligned}$$

Similarly to the sets of accepted and refused traces in CTG, the final states of a test case induce the sets of traces leading to a Pass (Inconc or Fail) verdict, denoted $[\![\mathfrak{M}_{TC}]\!]_{Pass}$ ($[\![\mathfrak{M}_{TC}]\!]_{Inconc}$ or $[\![\mathfrak{M}_{TC}]\!]_{Fail}$).

A test case is *controllable*, if the IUT has no chance anymore to choose between several outputs or between inputs and outputs, i.e.

$$\forall \sigma \in \Sigma^{TC} : (!\exists \xrightarrow{\lambda} : \lambda \in \Lambda^{TC}_{O}) \vee (\exists \xrightarrow{\lambda} : \lambda \in \Lambda^{TC}_{I} \wedge \neg \exists \xrightarrow{\lambda} : \lambda \in \Lambda^{TC}_{O}).$$

■

The test cases we treat in this chapter, are loopfree and controllable. They are executed in parallel with an IUT. The traces in a test case are chosen in a way that one trace leads to a Pass state. From this trace, several branches lead to Inconc states in one step. These Inconc states represent traces in the test purpose which end in a refusing state.

Definition 4.15 (Soundness; Rusu et al., 2000). Given a specification $\mathfrak{M}$, a test purpose $\mathfrak{M}_{TP}$ and a test case $\mathfrak{M}_{TC}$, a *verdict* of a test case $\mathfrak{M}_{TC}$ is *sound* if and only if the following holds for the executed trace π:

$$\text{setverdict}(\pi) = \begin{cases} \text{Pass} & \Leftrightarrow \pi \in [\![\mathfrak{M}_{TP}]\!]_{atrace} \cap [\![\mathfrak{M}]\!]_{traces} \\ \text{Inconc} & \Leftrightarrow \pi \in [\![\mathfrak{M}_{TP}]\!]_{rtrace} \cap [\![\mathfrak{M}]\!]_{traces} \\ \text{Fail} & \Leftrightarrow \pi \notin [\![\mathfrak{M}]\!]_{traces} \end{cases}$$

■

The verdict None is not considered here, since it is a construct to assign a test verdict to a trace that has not yet been executed. If for all traces in a test case a sound verdict is assigned, the test case is sound.

Using test purposes as selection criteria, it is possible to generate test cases on-the-fly without generating the whole state space of a specification. However, a CTG can easily be too large or even infinite, due to all possible data.

4.3 Chaotic Data Abstraction

Chaotic data abstraction has first been proposed by Sidorova and Steffen (2001b). They assume that a system is embedded into a chaotic environment, from which it can receive signals carrying *any* value. This can easily boost a system's state space to infinity, which makes the application of abstraction techniques mandatory in order to be able to analyze the system.

While most abstraction techniques make assumptions about those values from the environment by, for instance, dividing them into equivalence classes, the approach proposed by Sidorova and Steffen does not make any assumptions. This means, one can conceptually abstract values influenced by the environment via inputs and assignments to one abstract value, denoted $\mathbb{T}$ (*chaos*). That basically means ignoring these values and focusing on the control structure of a process.

System-internal data can be divided into two classes: Values that are not influenced by the environment remain the original ones, and so they should be treated in the same way as in the original system. Directly or indirectly influenced values are all transformed to the constant $\mathbb{T}$. For guards in the system, this leads to a three-valued logic, as we will discuss later. Sidorova and Steffen (2001b) and Ioustinova et al. (2004) proposed an approach to transform this three-valued logic back to the standard two-valued one, preserving in the abstracted system *at least* the behavior of the concrete one.

In this chapter, we implement chaotic data abstraction as a transformation on the level of system specifications in μCRL. Abstraction on the level of specifications is well developed within the Abstract Interpretation framework (Cousot and Cousot, 1977; Dams, 1996; Dams et al., 1997). The program transformation implementing this data abstraction transforms the signature and the process definition. For each sort $\mathbb{S}$, we introduce a sort $\mathbb{S}^{\mathbb{T}}$ that consists of two constructors, $\mathbb{T}_{\mathbb{S}} :\rightarrow \mathbb{S}^{\mathbb{T}}$ and $\kappa : \mathbb{S} \rightarrow \mathbb{S}^{\mathbb{T}}$. The first constructor defines a $\mathbb{T}$ value of the sort. The constructor κ (*known*) lifts values of sort $\mathbb{S}$ to values of sort $\mathbb{S}^{\mathbb{T}}$. For each concrete mapping $m : \mathbb{S}_1 \times \ldots \times \mathbb{S}_n \rightarrow \mathbb{S}$, we define a mapping $m^{\mathbb{T}} : \mathbb{S}_1^{\mathbb{T}} \times \cdots \times \mathbb{S}_n^{\mathbb{T}} \rightarrow \mathbb{S}^{\mathbb{T}}$ mimicking the original one on the abstracted sorts. In the general case, mimicking is ensured by providing the following rewrite rules for each abstract mapping $m^{\mathbb{T}}$:

$$\begin{aligned} m^{\mathbb{T}}(\kappa(x_1), \ldots, \kappa(x_n)) &= \kappa(m(x_1, \ldots, x_n)) \\ m^{\mathbb{T}}(x_1, \ldots, x_n) &= \mathbb{T}_{\mathbb{S}} \Leftrightarrow \exists i \in \{1; \ldots; n\} : x_i = \mathbb{T}_{\mathbb{S}_i} \end{aligned}$$

$$\text{IV-a}\ \frac{\ell \xrightarrow{?s(x)} \hat{\ell} \in E}{\ell \xrightarrow{?s(\mathbb{T})} \xrightarrow{T \triangleright x := \mathbb{T}} \hat{\ell} \in E^{\mathbb{T}}}$$

$$\text{IV-b}\ \frac{\ell \xrightarrow{g \triangleright !s(e)} \hat{\ell} \in E}{\ell \xrightarrow{\gamma(g^{\mathbb{T}}) \triangleright !s(e^{\mathbb{T}})} \hat{\ell} \in E^{\mathbb{T}}}$$

$$\text{IV-c}\ \frac{\ell \xrightarrow{g \triangleright x := e} \hat{\ell} \in E}{\ell \xrightarrow{\gamma(g^{\mathbb{T}}) \triangleright x := e^{\mathbb{T}}} \hat{\ell} \in E^{\mathbb{T}}}$$

Table 4.1: Transformation of edges ($\mathfrak{S} \to \mathfrak{S}^{\mathbb{T}}$)

The transformation of the process specification consists in lifting all variables, expressions and guards to the new sorts. Each occurrence of a variable x of sort $\mathbb{S}$, is substituted by an occurrence of the variable $x^{\mathbb{T}}$ of sort $\mathbb{S}^{\mathbb{T}}$, where $\mathbb{S}^{\mathbb{T}}$ is a safe abstraction of sort $\mathbb{S}$. Each occurrence of an expression e of sort $\mathbb{S}$ is lifted to an expression $e^{\mathbb{T}}$ of sort $\mathbb{S}^{\mathbb{T}}$. Thereby, all the newly introduced symbols (constructors and rewrite rules) are used and replace the appropriate original ones.

Transformation of guards is similar to the transformation of expressions. Every occurrence of a guard g is lifted to a guard $g^{\mathbb{T}}$ of sort $\mathrm{Bool}^{\mathbb{T}}$. While transforming guards we have to ensure that the abstract system shows *at least* the behavior of the original system. Therefore, the guards evaluated to $\kappa(T)$ or $\kappa(F)$, behave like guards evaluated to T or F, resp., while guards evaluated to $\mathbb{T}$, behave as guards evaluated to T. We implement this by introducing an extra mapping $\gamma : \mathrm{Bool}^{\mathbb{T}} \to \mathrm{Bool}$ that evaluates to T whenever a guard is evaluated either to $\mathbb{T}$ or to $\kappa(T)$ and to F otherwise. To avoid introducing unnecessary nondeterminism, we apply a more refined transformation to the sort Bool. Its abstraction, sort $\mathrm{Bool}^{\mathbb{T}}$, is shown in Figure 4.3.

Definition 4.16 (May Semantics for Chaotic Guards). While a guard g is defined as a function $g : \mathbb{S} \to \mathrm{Bool}$, a chaotic guard is defined as a function $g^{\mathbb{T}} : \mathbb{S}^{\mathbb{T}} \to \mathrm{Bool}^{\mathbb{T}}$. To map this three value logic back to a two value logic, a *may*-function $\gamma : \mathrm{Bool}^{\mathbb{T}} \to \mathrm{Bool}$ is defined as follows:

$$\gamma(b) = \begin{cases} T \Leftrightarrow b = \kappa(T) \vee b = \mathbb{T} \\ F \Leftrightarrow b = \kappa(F) \end{cases}$$

■

After transforming the specification's signature by adding the *chaotic* counterparts as described above and lifting system variables, expressions using the constructor κ as described above and guards using the function γ (see Definition 4.16), we obtain a system that still can receive all possible values from the environment. The environment can influence data only via inputs.

Let $\mathfrak{S} = (L, Var, A, E, (\ell_{init}, \eta_{init}))$ be a specification, G be its set of guards and E be its set of action names. Let $\ell, \hat{\ell} \in L$, $x \in Var$, $e \in Exprs(Var)$, $v \in \mathbb{D}$ for $e : \mathbb{D}$ with $\mathbb{D}$

$$
\begin{array}{ll}
\textbf{sort} & \mathrm{Bool}^{\mathbb{T}} \\
\textbf{func} & \mathbb{T}_{\mathrm{Bool}} :\rightarrow \mathrm{Bool}^{\mathbb{T}} \\
 & \kappa_{\mathrm{Bool}} : \mathrm{Bool} \rightarrow \mathrm{Bool}^{\mathbb{T}} \\
\textbf{map} & \mathrm{and}^{\mathbb{T}} : \mathrm{Bool}^{\mathbb{T}} \times \mathrm{Bool}^{\mathbb{T}} \rightarrow \mathrm{Bool}^{\mathbb{T}} \\
 & \ldots \\
 & \gamma : \mathrm{Bool}^{\mathbb{T}} \rightarrow \mathrm{Bool} \\
\textbf{var} & b, b' : \mathrm{Bool} \\
\textbf{rew} & \mathrm{and}^{\mathbb{T}}(\kappa(b), \kappa(b')) = \kappa(\mathrm{and}(b, b')) \\
 & \mathrm{and}^{\mathbb{T}}(\mathbb{T}_{\mathrm{Bool}}, \kappa(F)) = \kappa(F) \\
 & \mathrm{and}^{\mathbb{T}}(\kappa(F), \mathbb{T}_{\mathrm{Bool}}) = \kappa(F) \\
 & \mathrm{and}^{\mathbb{T}}(\mathbb{T}_{\mathrm{Bool}}, \kappa(T)) = \mathbb{T}_{\mathrm{Bool}} \\
 & \mathrm{and}^{\mathbb{T}}(\kappa(T), \mathbb{T}_{\mathrm{Bool}}) = \mathbb{T}_{\mathrm{Bool}} \\
 & \mathrm{and}^{\mathbb{T}}(\mathbb{T}_{\mathrm{Bool}}, \mathbb{T}_{\mathrm{Bool}}) = \mathbb{T}_{\mathrm{Bool}} \\
 & \ldots \\
 & \gamma(\mathbb{T}_{\mathrm{Bool}}) = T \\
 & \gamma(\kappa(b)) = b
\end{array}
$$

Figure 4.3: Transformed sort $\mathrm{Bool}^{\mathbb{T}}$

a data domain. We transform this specification to $\mathfrak{S}^{\mathbb{T}} = (L, \mathrm{Var}^{\mathbb{T}}, A^{\mathbb{T}}, E^{\mathbb{T}}, (\ell_{\mathrm{init}}, \eta^{\mathbb{T}}_{\mathrm{init}}))$ according to the rules given in Table 4.1. Strictly speaking, we transform every input $\ell \xrightarrow{?s(x)} \hat{\ell}$ from the environment into an input of signal s parameterized by the $\mathbb{T}$-value of the proper sort followed by assigning this $\mathbb{T}$-value to the variable x (see Rule IV-a in Table 4.1). Assignments and outputs are treated w.r.t. the rules in Table 2.1. The semantics of the transformed system is given by the inference rules in Table 4.2.

$$
\text{IV-d}\ \frac{\ell \xrightarrow{?s(x)} \xrightarrow{T \triangleright x := \mathbb{T}} \hat{\ell} \in E^{\mathbb{T}}}{(\ell, \eta^{\mathbb{T}}) \xrightarrow{?s(\mathbb{T})} (\hat{\ell}, \eta^{\mathbb{T}}_{]x \mapsto \mathbb{T}]})}
$$

$$
\text{IV-e}\ \frac{\ell \xrightarrow{\gamma(g) \triangleright !s(e)} \hat{\ell} \qquad [\![\gamma(g)]\!]_\eta = T \qquad [\![e]\!]_\eta = v}{(\ell, \eta) \xrightarrow{!s(v)} (\hat{\ell}, \eta)}
$$

$$
\text{IV-f}\ \frac{\ell \xrightarrow{\gamma(g) \triangleright x := e} \hat{\ell} \qquad [\![\gamma(g)]\!]_\eta = T \qquad [\![e]\!]_\eta = v}{(\ell, \eta) \xrightarrow{\tau} (\hat{\ell}, \eta[x \mapsto v])}
$$

Table 4.2: Step-semantics of transformed edges ($\mathfrak{S}^{\mathbb{T}} \rightarrow \mathfrak{M}^{\mathbb{T}}$)

$$\begin{array}{ccc} \mathfrak{S} & \xrightarrow{\text{Tab. 4.1}} & \mathfrak{S}^{\mathbb{T}} \\ \text{Tab. 2.1}\downarrow & & \downarrow\text{Tab. 4.2} \\ \mathfrak{M} & \preceq_{\leqslant} & \mathfrak{M}^{\mathbb{T}} \end{array}$$

$\mathfrak{M}^{\mathbb{T}}$ can receive only $\mathbb{T}$ values from the environment, so the infinity of environmental data is collapsed into a single value. Basically, the transformed system shows at least the traces of the original system where data influenced by the environment are substituted by $\mathbb{T}$ values. This means, that $\mathfrak{M}^{\mathbb{T}}$ simulates $\mathfrak{M}$. Further, we give an overview of preservation results based on Ioustinova (2004); Ioustinova et al. (2004).

An automaton $\mathfrak{M}_2$ simulates another automaton $\mathfrak{M}_1$, $\mathfrak{M}_1 \preceq_{\leqslant} \mathfrak{M}_2$, if for every transition in $\mathfrak{M}_1$ there is a simulating transition in $\mathfrak{M}_2$. A simulation transition in $\mathfrak{M}_2$ starts in a state, which stands in a simulating relation to the starting state of the transition in $\mathfrak{M}_1$ and also ends in one. Furthermore, the action, this transition in $\mathfrak{M}_2$ is labeled with, must stand in a simulation relation to the label, the original transition in $\mathfrak{M}_1$ is labeled with. Simulation relations on the level of states and action labels will be discussed later in this section.

Definition 4.17 ($\leqslant$-Simulation). Let $\mathfrak{M}_1 = (\Sigma^1, \Lambda^1, \Delta^1, \sigma^1_{\text{init}})$ and $\mathfrak{M}_2 = (\Sigma^2, \Lambda^2, \Delta^2, \sigma^2_{\text{init}})$ be two IOLTSs. Let furthermore $\sigma_1, \hat{\sigma}_1 \in \Sigma^1$, $\sigma_2, \hat{\sigma}_2 \in \Sigma^2$, $\lambda_1 \in \Lambda^1$ and $\lambda_2 \in \Lambda^2$. $(\leqslant_a, \leqslant_b)$ is a simulation, if and only if $\forall \sigma_1, \hat{\sigma}_1, \sigma_2, \lambda_1 . \exists \hat{\sigma}_2, \lambda_2 :$ $\sigma_1 \leqslant_a \sigma_2 \wedge \sigma_1 \xrightarrow{\lambda_1} \hat{\sigma}_1 \Rightarrow (\lambda_1 \leqslant_b \lambda_2 \wedge \hat{\sigma}_1 \leqslant_a \hat{\sigma}_2 \wedge \sigma_2 \xrightarrow{\lambda_2} \hat{\sigma}_2)$. We write $\mathfrak{M}_1 \preceq_{\leqslant} \mathfrak{M}_2$ if there is such a relation between $\mathfrak{M}_1$ and $\mathfrak{M}_2$, also relating their initial states $\sigma^1_{\text{init}} \leqslant_a \sigma^2_{\text{init}}$. ■

The defined simulation relation exceeds standard simulation relations in the aspect, that it is a relation between two different kinds of components of a system, namely states and transitions. While transitivity is given, an aspect we do not want to prove here since it is not related to our work, we will define a concrete simulation relation below, for which the statement that any trace in the simulating automaton is also a trace in the simulated one does not immediately hold, since the simulating automaton shows an overapproximation of behavior (safe abstraction). This aspect will be adjusted in Section 4.4 by an appropriate selection of data values for testing.

The simulation relation is now defined for concrete and abstracted IOLTSs. Before relating the traces of the transformed system to traces of the original system, we define an order relation on the states and on the labels of the systems. To relate the states $L \times Val$ of the original system with states of the transformed system $L \times Val^{\mathbb{T}}$, we define the relation $\leqslant_S$ on states as $\leqslant_S : \Sigma \times \Sigma^{\mathbb{T}}$. A state in $\mathfrak{M}_2$ simulates a state in $\mathfrak{M}_1$, if its valuation is either identical to that of the state in $\mathfrak{M}_1$, or all differing data elements have the value $\mathbb{T}$.

Definition 4.18 (Relation $\leqslant_S$). Let $\sigma = (\ell, \eta)$ and $\sigma^{\mathbb{T}} = (\ell, \eta^{\mathbb{T}})$ be two states of the IOLTSs $\mathfrak{M}$ and $\mathfrak{M}^{\mathbb{T}}$ with specifications $\mathfrak{S}$ and $\mathfrak{S}^{\mathbb{T}}$. $\leqslant_S : \Sigma \times \Sigma^{\mathbb{T}}$ is defined as $\sigma \leqslant_S \sigma^{\mathbb{T}}$ if and only if $\forall x \in Var : [\![x]\!]_{\eta^{\mathbb{T}}} = \mathbb{T} \vee [\![x]\!]_{\eta^{\mathbb{T}}} = \kappa([\![x]\!]_{\eta})$. ■

To relate labels λ of the original system with the labels of the transformed system $\lambda^{\mathbb{T}}$, we define the relation $\leqslant_L: \Lambda \times \Lambda^{\mathbb{T}}$. An action label on a transition in $\mathfrak{M}_2$ simulates an action on a transition in $\mathfrak{M}_1$, if both have the same name and their parameters have either identical values or those values differing are $\mathbb{T}$ for $\mathfrak{M}_2$.

Definition 4.19 (Relation $\leqslant_L$). Let $\lambda \in \Lambda$ and $\lambda^{\mathbb{T}} \in \Lambda^{\mathbb{T}}$. Then $\lambda \leqslant_L \lambda^{\mathbb{T}}$ is defined as follows:

- $\tau \leqslant_L \tau$;
- $?s(v) \leqslant_L ?s(v')$ if and only if either $v' = \mathbb{T}$ or $v' = \kappa(v)$;
- $!s(v) \leqslant_L !s(v')$ if and only if either $v' = \mathbb{T}$ or $v' = \kappa(v)$.

■

Lemma 4.20. Let $\mathfrak{S}$ be a specification and $\mathfrak{S}^{\mathbb{T}}$ be a specification obtained from $\mathfrak{S}$ by the transformation defined in this section. Let $\mathfrak{M}$ and $\mathfrak{M}^{\mathbb{T}}$ be IOLTSs obtained from respectively $\mathfrak{S}$ and $\mathfrak{S}^{\mathbb{T}}$ by the rules in Table 2.1 or Table 4.2, resp. Then $\mathfrak{M} \preceq_{\leqslant} \mathfrak{M}^{\mathbb{T}}$ and $(\leqslant_S, \leqslant_L)$ is this simulation. ■

Proof. The lemma can be proven based on Definition 4.17. Let a specification $\mathfrak{S}$ and its abstracted counterpart $\mathfrak{S}^{\mathbb{T}}$ be given as follows:

$$\begin{aligned} \mathfrak{S} &= (L, Var, A, E, (\ell_{init}, \eta_{init})) \\ \mathfrak{S}^{\mathbb{T}} &= (L^{\mathbb{T}}, Var, A, E^{\mathbb{T}}, (\ell_{init}, \eta^{\mathbb{T}}_{init})) \end{aligned}$$

η_{init} is the initial valuation of all global variables in $\mathfrak{S}$. Analogously, $\eta^{\mathbb{T}}_{init}$ is the initial valuation of all global variables in $\mathfrak{S}^{\mathbb{T}}$ with some values possibly being set to $\mathbb{T}$. The semantics of the specifications are given by two IOLTSs $\mathfrak{M}$ and $\mathfrak{M}^{\mathbb{T}}$ defined as follows:

$$\begin{aligned} \mathfrak{M} &= (\Sigma, \Lambda, \Delta, \sigma_{init}) \\ \mathfrak{M}^{\mathbb{T}} &= (\Sigma^{\mathbb{T}}, \Lambda^{\mathbb{T}}, \Delta^{\mathbb{T}}, \sigma^{\mathbb{T}}_{init}) \end{aligned}$$

Furthermore, let $\sigma, \hat{\sigma} \in \Sigma$, $\sigma \xrightarrow{\lambda} \hat{\sigma}$ in the set of transitions and $\sigma^{\mathbb{T}} \in \Sigma^{\mathbb{T}}$ be given. We prove the lemma by first considering the relationship between the initial states of both systems and then considering the relation of an arbitrary step.

Initial step – $\sigma_{init} \leqslant_S \sigma^{\mathbb{T}}_{init}$: We consider the initial states first. $\sigma^{\mathbb{T}}_{init} = (\ell_{init}, \eta^{\mathbb{T}}_{init})$ is derived from $\sigma_{init} = (\ell_{init}, \eta_{init})$ by substituting either none or some or all variable values in η_{init} by $\mathbb{T}$, while the control location ℓ_{init} stays the same all three cases. Substituting values in this way leads to $\eta^{\mathbb{T}}_{init}$. The relation $\leqslant_S$ holds here. If $\eta^{\mathbb{T}}$ is initialized with the original values from $\mathfrak{M}$, $\forall x \in Var : [\![x]\!]_{\eta^{\mathbb{T}}_{init}} = \kappa([\![x]\!]_{\eta_{init}})$ holds. If all initialization values are set to $\mathbb{T}$, $\forall x \in Var : [\![x]\!]_{\eta_{init}} = \mathbb{T}$ holds. For a mixed valuation, Var can be divided into exactly two disjunct subsets for which the two conditions above hold as well so that Definition 4.18 is fully satisfied.

General step – $\lambda \leqslant_L \lambda^{\mathbb{T}} \wedge \sigma \leqslant_S \sigma^{\mathbb{T}}$: Now, we consider the general step. Assume that $\sigma \leqslant_S \sigma^{\mathbb{T}}$ and let $\sigma \xrightarrow{\lambda} \hat{\sigma}$ in $\mathfrak{M}$ be given. Under these conditions, we can prove that:

- $\exists \sigma^{\mathbb{T}} \xrightarrow{\lambda^{\mathbb{T}}} \hat{\sigma}^{\mathbb{T}}$ in the set of transitions of $\mathfrak{M}^{\mathbb{T}}$ such that $\lambda \leqslant_L \lambda^{\mathbb{T}}$ and
- $\hat{\sigma} \leqslant_S \hat{\sigma}^{\mathbb{T}}$.

$$\begin{array}{ccc} \sigma & \leqslant_S & \sigma^{\mathbb{T}} \\ \lambda \downarrow & \leqslant_L & \downarrow \lambda^{\mathbb{T}} \\ \hat{\sigma} & \leqslant_S & \hat{\sigma}^{\mathbb{T}} \end{array}$$

We have to prove that an appropriate transition $\sigma^{\mathbb{T}} \xrightarrow{\lambda^{\mathbb{T}}} \hat{\sigma}^{\mathbb{T}}$ is generated in $\mathfrak{M}^{\mathbb{T}}$ with $\hat{\sigma} \leqslant_S \hat{\sigma}^{\mathbb{T}}$. In order to do so, we have to distinguish three different cases, namely input actions, output actions and τ-steps. In all three cases, the action in $\mathfrak{M}$ starts in a state $\sigma = (\ell, \eta)$, that in $\mathfrak{M}^{\mathbb{T}}$ starts in a state $\sigma^{\mathbb{T}} = (\ell, \eta^{\mathbb{T}})$ with $\sigma \leqslant_S \sigma^{\mathbb{T}} \Leftrightarrow (\ell, \eta) \leqslant_S (\ell, \eta^{\mathbb{T}})$.

Input action: Let $\lambda = ?s(x)$. The semantics of the step $\ell \xrightarrow{?s(x)} \hat{\ell}$ is given for $\mathfrak{M}$ in Table 2.1 as $(\ell, \eta) \xrightarrow{?s(v)} (\hat{\ell}, \eta_{[x \mapsto v]})$. For $\mathfrak{M}^{\mathbb{T}}$, Rule IV-a transforms the step to $\ell \xrightarrow{?s(\mathbb{T}) \to \mathbb{T} \triangleright x := \mathbb{T}} \hat{\ell}$. Its step semantics, given as Rule IV-d, finally leads to $(\ell, \eta^{\mathbb{T}}) \xrightarrow{?s(\mathbb{T})} (\hat{\ell}, \eta^{\mathbb{T}}_{[x \mapsto \mathbb{T}]})$. It holds that $?s(v) \leqslant_L ?s(\mathbb{T})$, since $?s(v) \leqslant_L ?s(v')$ with $v' = \mathbb{T}$. Furthermore, $(\hat{\ell}, \eta_{[x \mapsto v]}) \leqslant_S (\hat{\ell}, \eta^{\mathbb{T}}_{[x \mapsto \mathbb{T}[}})$, since $\forall x \in Var : (\llbracket x \rrbracket_{\eta^{\mathbb{T}}} = \kappa(\llbracket x \rrbracket_\eta)) \vee (\llbracket x \rrbracket_{\eta^{\mathbb{T}}} = \mathbb{T})$. With $\hat{\sigma} = (\hat{\ell}, \eta_{[x \mapsto v]})$ and $\hat{\sigma}^{\mathbb{T}} = (\hat{\ell}, \eta^{\mathbb{T}}_{[x \mapsto \mathbb{T}]})$, this immediately leads to $\hat{\sigma} \leqslant_S \hat{\sigma}^{\mathbb{T}}$.

Output action: Let $\lambda = g \triangleright !s(e)$. The step semantics for $\ell \xrightarrow{g \triangleright !s(e)} \hat{\ell}$ given in Table 2.1 is $(\ell, \eta) \xrightarrow{!s(v)} (\ell, \eta)$ with $\llbracket e \rrbracket_\eta = v$ for $\mathfrak{M}$ and $(\ell, \eta^{\mathbb{T}}) \xrightarrow{\gamma(g^{\mathbb{T}}) \triangleright !s(e^{\mathbb{T}})} (\ell, \eta^{\mathbb{T}})$ for $\mathfrak{M}^{\mathbb{T}}$. The fact, that $\eta \leqslant \eta^{\mathbb{T}}$ holds has already been shown above. It is guaranteed that this step appears in $\mathfrak{M}^{\mathbb{T}}$, since $\forall g : g \Rightarrow \gamma(g^{\mathbb{T}}))$ per definition (Definition 4.16). As defined in Definition 4.19, $!s(v) \leqslant_L !s(v^{\mathbb{T}})$ holds for $v^{\mathbb{T}} = \kappa(v)$. If $v^{\mathbb{T}}$ is influenced by $\mathbb{T}$ in $\mathfrak{M}^{\mathbb{T}}$, this leads to $!s(v) \leqslant_L !s(\mathbb{T})$, which also holds as of Definition 4.19. $\hat{\sigma} \leqslant_S \hat{\sigma}^{\mathbb{T}}$ for the same reason as shown above; there has even been no change in the valuation η or $\eta^{\mathbb{T}}$.

τ-step: Let λ be an assignment $x := e$. The semantics of the step $\ell \xrightarrow{g \triangleright x := e} \hat{\ell}$ is given for $\mathfrak{M}$ in Table 2.1 as $(\ell, \eta) \xrightarrow{\tau} (\hat{\ell}, \eta_{[x \mapsto e[}})$. For $\mathfrak{M}^{\mathbb{T}}$, Rule IV-c transforms the step to $\ell \xrightarrow{\gamma(g) \triangleright x := e^{\mathbb{T}}} \hat{\ell}$. Its step semantics, given as Rule IV-f finally leads to $(\ell, \eta^{\mathbb{T}}) \xrightarrow{\tau} (\hat{\ell}, \eta^{\mathbb{T}}_{[x \mapsto e^{\mathbb{T}}]})$. It is trivial to show that $\tau \leqslant_L \tau$ holds, since we have equal actions here without any data parameters. Furthermore, $(\hat{\ell}, \eta_{[x \mapsto c]}) \leqslant_S (\hat{\ell}, \eta^{\mathbb{T}}_{[x \mapsto c^{\mathbb{T}}[}})$, since $\forall x \in Var : (\llbracket x \rrbracket_{\eta^{\mathbb{T}}} = \kappa(\llbracket x \rrbracket_\eta)) \vee (\llbracket x \rrbracket_{\eta^{\mathbb{T}}} = \mathbb{T})$. With $\hat{\sigma} = (\hat{\ell}, \eta_{[x \mapsto c[}})$ and $\hat{\sigma}^{\mathbb{T}} = (\hat{\ell}, \eta^{\mathbb{T}}_{[x \mapsto c^{\mathbb{T}}[}})$, this immediately leads to $\hat{\sigma} \leqslant_S \hat{\sigma}^{\mathbb{T}}$.

□

Using Lemma 4.20, it is easy to show that every trace of $\mathfrak{M}$ can be mimicked by a trace of $\mathfrak{M}^{\mathbb{T}}$. For the lemma, we have to define, that two pairs of states are in a simulating relation, if their single components are.

Definition 4.21. Let (σ^1, σ^2) be a state of $\Sigma_1 \times \Sigma_2$ and (σ^3, σ^4) be a state of $\Sigma_3 \times \Sigma_4$. Then the following holds:

$$(\sigma^1, \sigma^2) \leqslant_S (\sigma^3, \sigma^4) \Leftrightarrow \sigma^1 \leqslant_S \sigma^3 \wedge \sigma^2 \leqslant_S \sigma^4.$$

■

Lemma 4.22. For the synchonous product of automata $\mathfrak{M}_1$ and $\mathfrak{M}_2$ with $\mathfrak{M}_3$ holds:

$$\mathfrak{M}_1 \preceq_{\leqslant} \mathfrak{M}_2 \Rightarrow \mathfrak{M}_1 \times \mathfrak{M}_3 \preceq_{\leqslant} \mathfrak{M}_2 \times \mathfrak{M}_3$$

under the simulation relation $(\leqslant_S, \leqslant_L)$. ■

Proof. Let $\sigma^1, \hat{\sigma}^1 \in \Sigma^1$, $\sigma^2 \in \Sigma^2$ and $\sigma^3 \in \Sigma^3$ be arbitrary states in the automata's state spaces. We assume that $(\sigma^1, \sigma^3) \leqslant_S (\sigma^2, \sigma^3)$, since $\mathfrak{M}_1 \preceq_{\leqslant} \mathfrak{M}_2$ following Definition 4.21. Furthermore, we can assume (see the proof for Lemma 4.20) that $\hat{\sigma}^1 \leqslant_S \hat{\sigma}^2$, $\hat{\sigma}^2 \in \Sigma^2$. As of Definition 4.21, $(\hat{\sigma}^1, \hat{\sigma}^3) \leqslant_S (\hat{\sigma}^2, \hat{\sigma}^3)$ since both $\hat{\sigma}^1 \leqslant_S \hat{\sigma}^2$ and $\hat{\sigma}^3 \leqslant_S \hat{\sigma}^3$ (trivial).

$$\begin{array}{ccc} (\sigma^1, \sigma^3) & \leqslant_S & (\sigma^2, \sigma^3) \\ \lambda \downarrow & \leqslant_L & \downarrow \lambda^{\mathbb{T}} \\ (\hat{\sigma}^1, \hat{\sigma}^3) & \leqslant_S & (\hat{\sigma}^2, \hat{\sigma}^3) \end{array}$$

Now let there be $\sigma^1 \xrightarrow{\lambda}{}^1 \hat{\sigma}^1$, $\sigma^2 \xrightarrow{\lambda}{}^2 \hat{\sigma}^2$ and $\sigma^3 \xrightarrow{\lambda}{}^3 \hat{\sigma}^3$. Building the synchronous product over the automata leads to the transitions $(\sigma^1, \sigma^3) \xrightarrow{\lambda}{}^{1\times3} (\hat{\sigma}^1, \hat{\sigma}^3)$ and $(\sigma^2, \sigma^3) \xrightarrow{\lambda}{}^{2\times3} (\hat{\sigma}^2, \hat{\sigma}^3)$. Building the synchronous product over two of the automata does not change anything about the action λ under consideration, so if $\lambda \leqslant_L \lambda$ holds for $\mathfrak{M}_1$ and $\mathfrak{M}_2$, it will do the same for $\mathfrak{M}_1 \times \mathfrak{M}_3$ and $\mathfrak{M}_2 \times \mathfrak{M}_3$. □

Now, we define, that two traces are in a simulation relation, if they are of the same length and their action labels are pairwise in such a simulation relation.

Definition 4.23 ($\leqslant$-inclusion on traces). Let π_1 and π_2 be traces of IOLTSs $\mathfrak{M}_1$ and $\mathfrak{M}_2$. Trace π_2 $\leqslant$-includes π_1, written $\pi_1 \leqslant \pi_2$, if and only if $|\pi_1| = |\pi_2|$ and $\pi_{1_\lambda}[i+1] \leqslant_L \pi_{2_\lambda}[i+1]$ for all $i \in \{0; \ldots; |\pi_1|\}$. ■

Finally, we lift the notion of a simulation relation to the level of automata.

Definition 4.24 ($\leqslant$-inclusion on automata). The set of traces generated by IOLTS $\mathfrak{M}_2$ $\leqslant$-includes the set of traces of $\mathfrak{M}_1$, written as $[\![\mathfrak{M}_1]\!]_{\text{traces}} \subseteq_{\leqslant} [\![\mathfrak{M}_2]\!]_{\text{traces}}$, if and only if for every trace π_1 of $\mathfrak{M}_1$ there exists a trace π_2 in $\mathfrak{M}_2$ such that $\pi_1 \leqslant \pi_2$. ■

Lemma 4.25. Let $\mathfrak{M}_{TP}$ be a test purpose, $\mathfrak{M}_{SP}$ be a synchronous product of $\mathfrak{M}$ with $\mathfrak{M}_{TP}$, and $\mathfrak{M}_{SP}^{\mathbb{T}}$ be a synchronous product of $\mathfrak{M}^{\mathbb{T}}$ with $\mathfrak{M}_{TP}$. Then $[\![\mathfrak{M}_{SP}]\!]_{\text{atrace}} \subseteq_{\leqslant} [\![\mathfrak{M}_{SP}^{\mathbb{T}}]\!]_{\text{atrace}}$ and $[\![\mathfrak{M}_{SP}]\!]_{\text{rtrace}} \subseteq_{\leqslant} [\![\mathfrak{M}_{SP}^{\mathbb{T}}]\!]_{\text{rtrace}}$. ■

Proof. Let $\mathfrak{M} \times \mathfrak{M}_{TP}$ be the synchronous product of a system and a test purpose and let $\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}$ be the synchronous product of an abstracted system and the same test purpose. For both products, the sets of traces are defined as the disjoint sets of accepting traces, which end in an accepting state, and refusing traces ending in a refusing state. That means $[\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{traces} = [\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{atrace} \dot{\cup} [\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{rtrace}$ and $[\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{traces} = [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{atrace} \dot{\cup} [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{rtrace}$. We have to show that the following holds:

1. $[\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{atrace} \subseteq_{\leqslant} [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{atrace}$ and
2. $[\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{rtrace} \subseteq_{\leqslant} [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{rtrace}$.

As we have proved before, $\mathfrak{M} \times \mathfrak{M}_{TP} \preceq_{\leqslant} \mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}$ holds (Lemmata 4.20 and 4.22). It follows from Definitions 4.23 and 4.24 that $[\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{atrace} \dot{\cup} [\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{rtrace} \subseteq_{\leqslant} [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{atrace} \dot{\cup} [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{rtrace}$. From this we can conclude that

1. $[\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{atrace} \subseteq_{\leqslant} [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{atrace} \dot{\cup} [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{rtrace}$ and
2. $[\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{rtrace} \subseteq_{\leqslant} [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{atrace} \dot{\cup} [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{rtrace}$.

To prove the above claim, we have to show that

1. $[\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{atrace} \cap [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{rtrace} = \emptyset$ and
2. $[\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{rtrace} \cap [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{rtrace} = \emptyset$.

This is easy to show. There exists no trace $\pi \in [\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{atrace}$ which would ever end in a refusing state, so we can conclude immediately $\pi \in [\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{atrace} \Rightarrow \pi \notin [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{rtrace}$. Analogously, we can say that $\pi \in [\![\mathfrak{M} \times \mathfrak{M}_{TP}]\!]_{rtrace} \Rightarrow \pi \notin [\![\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}]\!]_{atrace}$. For this reason, our entrance claim holds. □

4.4 Parameterizing Test Cases with Data

In this section, we will complete the test generation approach with data abstraction from the previous section by discussing test case parameterization. Here, we will discuss a static approach for the parameterization of controllable test cases (see Definition 4.14). In Chapter 5, we will present a holistic approach to behavior-adapting test case execution with test data selection, which is founded on CTGs (Definition 4.12).

Our goal is to obtain parameterizable test cases, for instance in TTCN-3 as defined by the ETSI (2003a), together with information about values that can be used to instantiate them. In a first step towards the full functionality of BAiT (cf. Chapter 5), we are interested in test cases where no nondeterministic choice is possible between several outputs or between inputs and outputs. Therefore, we single out a subgraph of $\mathfrak{M}^{\mathbb{T}}_{CTG}$ that contain neither choices between several outputs or choices between inputs and outputs nor loops. We refer further to this subgraph as an Abstract Test Case (ATC), denoted $\mathfrak{M}^{\mathbb{T}}_{TC}$.

Even though we are still working on the level of IOLTSs here, we now have to introduce variables for parameterization. In $\mathfrak{M}^{\mathbb{T}}_{TC}$, each occurrence of $\mathbb{T}$ is substituted

by a unique symbolic variable v_{i_j} parameterizing inputs and outputs. The double index is necessary to identify the state in which the transition with the variable starts (index i), and to uniquely identify this variable within the set of variables on transitions from state i (index j). These variables are embedded into the transition labels of the IOLTS, but are distinguished as a separate set Var in the further regard. A parameterizable test case is thus in principle a CTG with variable parameters on actions.

Definition 4.26 (Parameterizable Test Case). Given a CTG

$$\mathfrak{M}^{\mathbb{T}}_{CTG} = (L \times Val, Var_{CTG}, \Lambda, \Delta_{CTG}, (\ell_{init}, \eta_{init})),$$

a *parameterizable test case*

$$\mathfrak{M}^{\mathbb{T}}_{TC} = (L' \times Val, Var_{TC}, \Lambda, \Delta_{TC}, (\ell_{init}, \eta_{init}))$$

is an input complete IOLTS, where $Var_{TC} \subseteq Var_{CTG}$, $L' \times Val \subseteq L \times Val$, and the test case shows only Pass, Inconc and Fail traces possible in the complete test graph, i.e. $[\![\mathfrak{M}^{\mathbb{T}}_{TC}]\!]_{Pass} \subseteq [\![\mathfrak{M}^{\mathbb{T}}_{CTG}]\!]_{Pass}$, $[\![\mathfrak{M}^{\mathbb{T}}_{TC}]\!]_{Inconc} \subseteq [\![\mathfrak{M}^{\mathbb{T}}_{CTG}]\!]_{Inconc}$, and $[\![\mathfrak{M}^{\mathbb{T}}_{TC}]\!]_{Fail} \subseteq [\![\mathfrak{M}^{\mathbb{T}}_{CTG}]\!]_{Fail}$. ■

Before such a test case can be executed, it must be instantiated. This means, that each of the variables $v_{i_j} \in Var$ must be preset with a value such that a Pass-state in the test case can be reached with this valuation η_{init}. In order to do so, a trace to Pass is selected, transformed into the test oracle, and by constraint solving the appropriate test case parameters are determined. The selection of a trace to Pass is in case of a controllable test case already taken by TGV. In case of a CTG, we will discuss the according algorithms in Chapter 5.

Building the Test Oracle

A parameterizable test case may contain traces introduced by data abstraction. Moreover, information about the relationship of symbolic variables or concrete values with which they can be substituted is absent. To sort out spurious traces and to obtain information about valuations for symbolic variables, we employ constraint solving.

In blackbox testing, the idea of *test oracles* has been established. As of the ISTQB (2006), a test oracle is "a source to determine expected results to compare with the actual result of the software under test". The IUT receives input from the tester and returns output to the tester. This output is then compared to expected results, which are delivered by the test oracle. The test oracle in turn bases its decisions on sources like the specification of the IUT.

In our approach, we slightly extend the meaning of *test oracle* by also letting it compute possible inputs to the IUT, as far as those can be determined by the constraint solver. Where this is not possible, we have to apply other test data selection techniques. A test oracle is a CLP, which has been generated from the data definitions and the process definitions of a μCRL specification according to Sections 3.2 and 3.3.

Definition 4.27 (Test Oracle). Given a CLP $\mathfrak{P} = (\mathfrak{P}_{ADT}, \mathfrak{P}_{Proc})$ of a specification $\mathfrak{S}$, an ATC $\mathfrak{M}_{TC}^{\mathbb{T}}$ and a trace $\pi \in [\![\mathfrak{M}_{TC}^{\mathbb{T}}]\!]_{traces}$ which leads to a Pass verdict. Then $\mathfrak{O}_\pi(\theta) = (\mathfrak{P}, q_\pi(\theta))$ is a test oracle over $\mathfrak{P}$ and a trace-dependent query $q_\pi(\theta)$ with solution θ. ■

Remark 4.28. For a test oracle, we define a query for the selected trace as a function within the test oracle. While it is explicitly part of the test oracle in this section, we will dynamically create this query during test execution in Chapter 5. ■

In order to generate the oracle, we generate a CLP $\mathfrak{P}$ as described in Chapter 3. This CLP can then be queried. The Pass-trace π, which is selected from $\mathfrak{M}_{TC}^{\mathbb{T}}$, is transformed into a query $q_\pi(\theta)$. Let the set of symbolic variables in the specification be Var_{symb}. If there is a solution $\theta : Var_{symb} \to \mathbb{D}^*$ in $\mathfrak{P}$ for the query, the trace π can be mapped to a trace of the original system.

We refer to trace π with symbolic variables substituted according to θ as an *instantiated trace* denoted $\pi(\theta)$. The instantiated trace $\pi(\theta)$ is a trace of the original system $\mathfrak{M}$. As we will prove later in Chapter 5 for general test execution, the verdict assigned to a test run of $\pi(\theta)$ is sound.

$$
query(i, r) = \begin{cases}
[] \iff r = [] \\
\left[\tau(state(\ell_{init}, \overline{Var}_{init}), state(\ell_1, \overline{Var}_1), param) | query(1, r')\right], \\
\iff i = 0 \wedge r = \left[\sigma \xrightarrow{\tau} \sigma' | r'\right] \\
\left[s(state(\ell_{init}, \overline{Var}_{init}), state(\ell_1, \overline{Var}_1), param(y)) | query(1, r')\right], \\
\iff i = 0 \wedge \left(r = \left[\sigma \xrightarrow{?s(y)} \sigma' | r'\right] \vee r = \left[\sigma \xrightarrow{!s(y)} \sigma' | r'\right]\right) \\
\left[\tau(state(\ell_i, \overline{Var}_i), state(\ell_{i+1}, \overline{Var}_{i+1}), param) | query(i+1, r')\right], \\
\iff i > 0 \wedge r = \left[\sigma \xrightarrow{\tau} \sigma' | r'\right] \\
\left[s(state(\ell_i, \overline{Var}_i), state(\ell_{i+1}, \overline{Var}_{i+1}), param(y)) | query(i+1, r')\right], \\
\iff i > 0 \wedge \left(r = \left[\sigma \xrightarrow{?s(y)} \sigma' | r'\right] \vee r = \left[\sigma \xrightarrow{!s(y)} \sigma' | r'\right]\right)
\end{cases}
$$

Figure 4.4: Transformation of a trace of $\pi \in [\![\mathfrak{M}_{TC}^{\mathbb{T}}]\!]_{traces}$ into a query q_π

In order to query the oracle, we transform the Pass-trace π into a query $q_\pi := query(0, \pi)$ using the function given in Figure 4.4. Basically, a query is a sequence of rule invocations corresponding to the transitions along the chosen Pass-trace. Each transition along the trace is transformed into a rule invocation, which has the name of the action under consideration. The parameters of this rule invocation are the state of the system where the transition starts (first parameter), the system's state after the transition and the action's parameters. In the first transition, which is characterized by the counter $i = 0$, the starting state of the transition is set to the initial state of the system. The function query then iterates through the trace and appends all rule invocations to one list, which forms the oracle.

In q_π, not all free variables in the system have yet been bound to values. This happens by applying the constraint solver to the test oracle with $\theta := solve(\mathfrak{P}, q_\pi, \theta_{const})$.

Sometimes a trace has not yet been fully instantiated, i.e. some values for data elements have already been defined while others have not. Such a trace is a *partially valuated* trace.

Definition 4.29 (Partial Valuation). Let $vars : [\![\mathfrak{M}]\!]_{traces} \rightarrow Var$ be a function that projects the set of variables Var of $\mathfrak{M}$ to that subset that is actually used in a given trace from $[\![\mathfrak{M}]\!]_{traces}$.

Given a valuation $\theta : vars(\pi) \rightarrow \mathbb{D}^*$ and a trace β, which is a prefix of π, we define the *partial valuation* $\lfloor\theta\rfloor_\beta : vars(\beta) \rightarrow \mathbb{D}^*$ such that $\lfloor\theta\rfloor_\beta(x) = \theta(x) \forall x \in vars(\beta)$. ■

The parameter $\beta_{const} \subseteq \beta$ can be used to define a set of constant valuation assignments. For instance, if a prefix β of π has already been executed during a test and only for the suffix of π a new valuation has to be found, $\theta_{const} := \lfloor\theta\rfloor_\beta$ can be defined as this set of constant values. Partial valuation and constant valuation assignments in the context of trace prefixed will become important in Chapter 5.

In all cases, where this situation is not applicable, i.e. no part of θ has to be constant, the optional parameter θ_{const} can be defined as $\emptyset$ and is further ignored. Having calculated a valuation θ for a trace π, the constraint solver can check, whether $\langle q_\pi(\theta), \top\rangle$ is solvable. If it is solvable, the test can be executed with the computed test input data, and results from the IUT can be checked against the computed test output data.

Lemma 4.30. Let $\pi(\theta)$ be a trace π of the ATC for the system specified by $\mathfrak{M}$. Let π be instantiated with the valuation θ. Then:

$$\mathfrak{P} \vdash q_\pi(\theta) \Leftrightarrow \pi(\theta) \in [\![\mathfrak{M}]\!]_{traces}$$

Proof. To show the equivalence stated above, we have to divide the proof into two parts, proving each direction separately.

$\mathfrak{P} \vdash q_\pi(\theta) \Leftarrow \pi(\theta) \in [\![\mathfrak{M}]\!]_{traces}$ We begin with proving the lemma for test queries consisting of only one step.

1. Let $\pi(\theta)$ be a trace of *one* transition

$$\sigma_{init} \xrightarrow{\tau} \hat{\sigma} \equiv (\ell_{init}, \overline{Var}_{init}) \xrightarrow{\tau} (\hat{\ell}, \overline{Var}_{init[x \mapsto v]}).$$

As defined in the step semantics in Table 2.1 (Rule II-c), the appropriate step in the specification is $\ell_{init} \xrightarrow{g \triangleright x := e} \hat{\ell}$ with $[\![g]\!]_\theta = \top$ and $v = [\![e]\!]_\theta$. According to Rule III-w in Table 3.1, the CLP $\mathfrak{P}$ contains the rule

$$\tau(state(\ell_{init}, \overline{Var}_{init}), state(\hat{\ell}, \overline{Var}_{init[x \mapsto e]}), param) \leftarrow g.$$

Following the function query in Figure 4.4, the query q_π only contains the rule invocation

$$\tau(state(\ell_{init}, \overline{Var}_{init}), state(\ell_1, \overline{Var}_1), param).$$

This query holds since

a) $\mathfrak{P}$ contains the appropriate rule (see above),

b) this rule instantiates e with $[\![e]\!]_\theta$ when invoked, and

c) this rule holds for $\hat{\ell} = \ell_1$, $\overline{Var}_{init[x \mapsto e[} = \overline{Var}_1$ under valuation θ and $[\![g]\!]_\theta = \top$.

2. Let $\pi(\theta)$ be a trace of *one* transition

$$\sigma_{init} \xrightarrow{!s(v)} \hat{\sigma} \equiv (\ell_{init}, \overline{Var}_{init}) \xrightarrow{!s(v)} (\hat{\ell}, \overline{Var}_{init}).$$

As defined in the step semantics in Table 2.1 (Rule II-b), the appropriate step in the specification is $\ell \xrightarrow{g \triangleright !s(e)} \hat{\ell}$ with $[\![g]\!]_\theta = \top$ and $v = [\![e]\!]_\theta$. According to Rule III-u in Table 3.1, the CLP $\mathfrak{P}$ contains the rule

$$s(state(\ell_{init}, \overline{Var}_{init}), state(\hat{\ell}, \overline{Var}_{init}), param(e)) \leftarrow g.$$

Following the function *query* in Figure 4.4, the query q_π only contains the rule invocation

$$s(state(\ell_{init}, \overline{Var}_{init}), state(\ell_1, \overline{Var}_1), param(v))$$

with $v = [\![e]\!]_\theta$. This query holds since

a) $\mathfrak{P}$ contains the appropriate rule (see above),

b) this rule instantiates e with $[\![e]\!]_\theta$ when invoked, and

c) this rule holds for $\hat{\ell} = \ell_1$, $\overline{Var}_{init} = \overline{Var}_1$ under valuation θ and $[\![g]\!]_\theta = \top$.

3. Let $\pi(\theta)$ be a trace of *one* transition

$$\sigma_{init} \xrightarrow{?s(v)} \hat{\sigma} \equiv (\ell_{init}, \overline{Var}_{init}) \xrightarrow{?s(v)} (\hat{\ell}, \overline{Var}_{init[x \mapsto v]}).$$

As defined in the step semantics in Table 2.1 (Rule II-a), the appropriate step in the specification is $\ell_{init} \xrightarrow{?s(x)} \hat{\ell}$. According to Rule III-v in Table 3.1, the CLP $\mathfrak{P}$ contains the rule

$$s(state(\ell_{init}, \overline{Var}_{init}), state(\hat{\ell}, \overline{Var}_{init[x \mapsto y]}), param(y)).$$

Following the function *query* in Figure 4.4, the query q_π only contains the rule invocation

$$s(state(\ell_{init}, \overline{Var}_{init}), state(\ell_1, \overline{Var}_1), param(v)).$$

Thus, this query holds since

a) $\mathfrak{P}$ contains the appropriate rule (see above),

b) this rule instantiates y with v when invoked, and

c) this rule holds for $\hat{\ell} = \ell_1$ and $\overline{Var}_{[x \mapsto y]} = \overline{Var}_1$ under valuation θ.

Since we assume the constraint-solver to work correctly, this will be the case if $\pi(\theta) \in [\![\mathfrak{M}]\!]_{\text{traces}}$.

Now we regard the general step of the proof. We assume to have an query q_β, which holds under θ since $\beta \in [\![\mathfrak{M}]\!]_{\text{traces}}$. We extend β by one transition to completely describe trace $\pi \in [\![\mathfrak{M}]\!]_{\text{traces}}$ with the prefix β.

4. Let $\pi(\theta)$ be a trace with the prefix β followed by the transition

$$\sigma \xrightarrow{\tau} \hat{\sigma} \equiv (\ell, \overline{Var}) \xrightarrow{\tau} (\hat{\ell}, \overline{Var}_{[x \mapsto v]}).$$

   As defined in the step semantics in Table 2.1 (Rule II-c), the appropriate step in the specification is $\ell \xrightarrow{g \triangleright x := e} \hat{\ell}$ with $[\![g]\!]_\theta = \top$ and $v = [\![e]\!]_\theta$. According to Rule III-w in Table 3.1, the CLP $\mathfrak{P}$ contains the rule

$$\tau(state(\ell, \overline{Var}), state(\hat{\ell}, \overline{Var}_{[x \mapsto c]}), param) \leftarrow g.$$

   Following the function query in Figure 4.4, the query q_π only contains the rule invocation

$$\tau(state(\ell_n, \overline{Var}_n), state(\ell_{n+1}, \overline{Var}_{n+1}), param).$$

   This query holds since

   a) $\mathfrak{P}$ contains the appropriate rule (see above),
   b) this rule instantiates e with $[\![e]\!]_\theta$ when invoked, and
   c) this rule holds for $l = \ell_n$, $\hat{\ell} = \ell_{n+1}$, $\overline{Var} = \overline{Var}_n$, $\overline{Var}_{[x \mapsto c]} = \overline{Var}_{n+1}$ under valuation θ and $[\![g]\!]_\theta = \top$.

5. Let $\pi(\theta)$ be a trace with the prefix β followed by the transition

$$\sigma \xrightarrow{!s(v)} \hat{\sigma} \equiv (\ell, \overline{Var}) \xrightarrow{!s(v)} (\hat{\ell}, \overline{Var}).$$

   As defined in the step semantics in Table 2.1 (Rule II-b), the appropriate step in the specification is $\ell \xrightarrow{g \triangleright !s(e)} \hat{\ell}$ with $[\![g]\!]_\theta = \top$ and $v = [\![e]\!]_\theta$. According to Rule III-u in Table 3.1, the CLP $\mathfrak{P}$ contains the rule

$$s(state(\ell, \overline{Var}), state(\hat{\ell}, \overline{Var}), param(e)) \leftarrow g.$$

   Following the function query in Figure 4.4, the query q_π only contains the rule invocation

$$s(state(\ell_n, \overline{Var}_n), state(\ell_{n+1}, \overline{Var}_{n+1}), param(v))$$

   with $v = [\![e]\!]_\theta$. This query holds since

   a) $\mathfrak{P}$ contains the appropriate rule (see above),
   b) this rule instantiates e with $[\![e]\!]_\theta$ when invoked, and

c) this rule holds for $l = \ell_n$, $\hat{\ell} = \ell_{n+1}$, $\overline{Var} = \overline{Var}_n = \overline{Var}_{n+1}$ under valuation θ and $[\![g]\!]_\theta = \top$.

6. Let $\pi(\theta)$ be a trace with the prefix β followed by the transition

$$\sigma \xrightarrow{?s(v)} \hat{\sigma} \equiv (\ell, \overline{Var}) \xrightarrow{?s(v)} (\hat{\ell}, \overline{Var}_{[x \mapsto v]}).$$

As defined in the step semantics in Table 2.1 (Rule II-a), the appropriate step in the specification is $\ell \xrightarrow{?s(x)} \hat{\ell}$. According to Rule III-v in Table 3.1, the CLP $\mathfrak{P}$ contains the rule

$$s(state(\ell, \overline{Var}, state(\hat{\ell}, \overline{Var}_{[x \mapsto y]}), param(y)).$$

Following the function query in Figure 4.4, the query q_π only contains the rule invocation

$$s(state(\ell_n, \overline{Var}_n), state(\ell_{n+1}, \overline{Var}_{n+1}), param(v)).$$

Thus, this query holds since

a) $\mathfrak{P}$ contains the appropriate rule (see above),

b) this rule instantiates y with v, when invoked, and

c) this rule holds for $l = \ell_n$, $\hat{\ell} = \ell_{n+1}$, $\overline{Var} = \overline{Var}_n$ and $\overline{Var}_{[x \mapsto y]} = \overline{Var}_{n+1}$ under valuation θ.

Since we assume the constraint-solver to work correctly, this will again be the case if $\pi(\theta) \in [\![\mathfrak{M}]\!]_{traces}$.

$\underline{\mathfrak{P} \vdash q_\pi(\theta) \Rightarrow \pi(\theta) \in [\![\mathfrak{M}]\!]_{traces}}$ We begin with the trace consisting of one transition again.

1. τ *Step:* Let q_π contain only the rule invocation

$$\tau(state(\ell_{init}, \overline{Var}_{init}), state(\ell_1, \overline{Var}_1), param).$$

This is the final situation after having applied the function query to a trace π, which contains one transition

$$\sigma_{init} \xrightarrow{\tau} \hat{\sigma} \equiv (\ell_{init}, \overline{Var}_{init}) \xrightarrow{\tau} (\hat{\ell}, \overline{Var}_{init[x \mapsto v]})$$

only (see Figure 4.4). The transition itself has been generated from a specification with an edge $\ell_{init} \xrightarrow{g \triangleright x := e} \hat{\ell}$, $[\![g]\!]_\theta = \top$ and $[\![e]\!]_\theta = v$ (see Table 2.1, Rule II-c). q_π holds under θ because of the rule

$$\tau(state(\ell_{init}, \overline{Var}_{init}), state(\hat{\ell}, \overline{Var}_{init[x \mapsto v]}), param) \leftarrow g$$

in $\mathfrak{P}$, which holds under θ with $v = [\![e]\!]_\theta$, $\hat{\ell} = \ell_1$ and $\overline{Var}_{init[x \mapsto v]} = \overline{Var}_1$. This rule has been generated from $\mathfrak{M}$ by III-w (see Table 3.1) for the named edge.

2. *Output action:* Let q_π contain only the rule invocation

$$s(state(\ell_{init}, \overline{Var}_{init}), state(\ell_1, \overline{Var}_1), param(v))$$

with $v = [\![e]\!]_\theta$. This is the final situation after having applied the function query to a trace π, which contains one transition

$$\sigma_{init} \xrightarrow{!s(v)} \hat{\sigma} \equiv (\ell_{init}, \overline{Var}_{init}) \xrightarrow{!s(v)} (\hat{\ell}, \overline{Var}_{init})$$

only (see Figure 4.4). The transition itself has been generated from a specification with an edge $\ell_{init} \xrightarrow{g \triangleright !s(e)} \hat{\ell}$, $[\![g]\!]_\theta = \top$ and $[\![e]\!]_\theta = v$ (see Table 2.1, Rule II-b). q_π holds under θ because of the rule

$$s(state(\ell_{init}, \overline{Var}_{init}), state(\hat{\ell}, \overline{Var}_{init}), param(e)) \leftarrow g$$

in $\mathfrak{P}$, which holds under θ with e being instantiated with $[\![e]\!]_\theta$, $v = [\![e]\!]_\theta$, $\hat{\ell} = \ell_1$ and $\overline{Var}_{init} = \overline{Var}_1$. This rule has been generated from $\mathfrak{M}$ by III-u (see Table 3.1) for the named edge.

3. *Input action:* Let q_π contain only the rule invocation

$$s(state(\ell_{init}, \overline{Var}_{init}), state(\ell_1, \overline{Var}_1), param(v)).$$

This is the final situation after having applied the function query to a trace π, which contains one transition

$$\sigma_{init} \xrightarrow{?s(v)} \hat{\sigma} \equiv (\ell_{init}, \overline{Var}_{init}) \xrightarrow{?s(v)} (\hat{\ell}, \overline{Var}_{init[x \mapsto v]})$$

only (see Figure 4.4). The transition itself has been generated from a specification with an edge $\ell_{init} \xrightarrow{?s(x)} \hat{\ell}$ (see Table 2.1, Rule II-a). q_π holds under θ because of the rule

$$s(state(\ell_{init}, \overline{Var}_{init}), state(\hat{\ell}, \overline{Var}_{init[x \mapsto y]}), param(y))$$

in $\mathfrak{P}$, which holds under θ with $y = v$, $\hat{\ell} = \ell_1$ and $\overline{Var}_{init[x \mapsto v]} = \overline{Var}_1$. This rule has been generated from $\mathfrak{M}$ by III-v (see Table 3.1) for the named edge.

Since we only regard a one-transition trace π here, $\pi \in [\![\mathfrak{M}]\!]_{traces}$ holds in all three cases.

Now we assume an query q_β which holds under θ since there exists a $\pi(\theta) \in [\![\mathfrak{M}]\!]_{traces}$. In the general step, we add a rule invocation to the query to complete q_π for trace π with its prefix β. We then show that $\pi(\theta) \in [\![\mathfrak{M}]\!]_{traces}$.

4. τ *Step:* Let q_π contain the rule invocations for q_β and one additional rule invocation

$$\tau(state(\ell_n, \overline{Var}_n), state(\ell_{n+1}, \overline{Var}_{n+1}), param).$$

This is the final situation after having applied the function query to a trace π, which contains the transition

$$\sigma \xrightarrow{\tau} \hat{\sigma} \equiv (\ell, \overline{Var}) \xrightarrow{\tau} (\hat{\ell}, \overline{Var}_{[x \mapsto v]})$$

only (see Figure 4.4). The transition itself has been generated from a specification with an edge $\ell \xrightarrow{g \triangleright x := e} \hat{\ell}$, $[\![g]\!]_\theta = \top$ and $[\![e]\!]_\theta = v$ (see Table 2.1, Rule II-c). q_π holds under θ because of the rule

$$\tau(state(\ell, \overline{Var}), state(\hat{\ell}, \overline{Var}_{[x \mapsto v]}), param) \leftarrow g$$

in $\mathfrak{P}$, which holds under θ with $v = [\![e]\!]_\theta$, $l = \ell_n$, $\hat{\ell} = \ell_{n+1}$, $\overline{Var} = \overline{Var}_n$ and $\overline{Var}_{[x \mapsto v]} = \overline{Var}_{n+1}$. This rule has been generated from $\mathfrak{M}$ by III-w (see Table 3.1) for the named edge.

5. *Output action:* Let q_π contain only the rule invocation

$$s(state(\ell_n, \overline{Var}_n), state(\ell_{n+1}, \overline{Var}_{n+1}), param(v))$$

with $v = [\![e]\!]_\theta$. This is the final situation after having applied the function query to a trace π, which contains one transition

$$\sigma \xrightarrow{!s(v)} \hat{\sigma} \equiv (\ell, \overline{Var}) \xrightarrow{!s(v)} (\hat{\ell}, \overline{Var})$$

only (see Figure 4.4). The transition itself has been generated from a specification with an edge $\ell_{init} \xrightarrow{g \triangleright !s(e)} \hat{\ell}$, $[\![g]\!]_\theta = \top$ and $[\![e]\!]_\theta = v$ (see Table 2.1, Rule II-b). q_π holds under θ because of the rule

$$s(state(\ell, \overline{Var}), state(\hat{\ell}, \overline{Var}), param(e)) \leftarrow g$$

in $\mathfrak{P}$, which holds under θ with e being instantiated with $[\![e]\!]_\theta$, $v = [\![e]\!]_\theta$, $l = \ell_n$ $\hat{\ell} = \ell_{n+1}$ and $\overline{Var} = \overline{Var}_n = \overline{Var}_{n+1}$. This rule has been generated from $\mathfrak{M}$ by III-u (see Table 3.1) for the named edge.

6. *Input action:* Let q_π contain only the rule invocation

$$s(state(\ell_n, \overline{Var}_n), state(\ell_{n+1}, \overline{Var}_{n+1}), param(v)).$$

This is the final situation after having applied the function query to a trace π, which contains one transition

$$\sigma \xrightarrow{?s(v)} \hat{\sigma} \equiv (\ell, \overline{Var}) \xrightarrow{?s(v)} (\hat{\ell}, \overline{Var}_{[x \mapsto v]})$$

only (see Figure 4.4). The transition itself has been generated from a specification with an edge $\ell \xrightarrow{?s(x)} \hat{\ell}$ (see Table 2.1, Rule II-a). q_π holds under θ because of the rule

$$s(state(\ell, \overline{Var}), state(\hat{\ell}, \overline{Var}_{[x \mapsto y]}), param(y))$$

in $\mathfrak{P}$, which holds under θ with $y = v$, $l = \ell_n$, $\hat{\ell} = \ell_{n+1}$, $\overline{Var} = \overline{Var}_n$ and $\overline{Var}_{[x \mapsto v]} = \overline{Var}_{n+1}$. This rule has been generated from $\mathfrak{M}$ by III-v (see Table 3.1) for the named edge.

Since the prefix β of π holds under θ and there exists one of the transitions named above, which also holds under θ, $\pi \in [\![\mathfrak{M}]\!]_{traces}$ holds. □

4.5 CEPS Case Study

In this section, we describe the application of our approach to the case study Common Electronic Purse Specifications (CEPS). The CEPS define a protocol for electronic payment using a chip card as a wallet. The specifications consist of the *functional requirements* (CEPSCO, 1999) and the *technical specification* (CEPSCO, 2000). A complete electronic purse system covers three roles: a card user, a card issuer (the issuing bank institute, for instance) and a card reader as a connection between these two. The hardware of such a system is given by the purse card itself, the card reader and some network infrastructure. Software applications are running on the card, the card reader and at the site of the card issuer. These applications communicate with each other. Their collaboration is depicted in Figure 4.5.

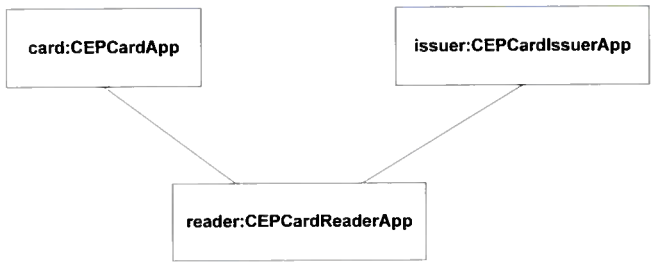

Figure 4.5: Collaborating system roles in CEPS

In our work on the case study, we aim to evaluate our test generation process by automatically generating parameterizable test cases from a μCRL specification for the card application `CEPCardApp` in Figure 4.5. Our approach starts from a formalized version[1] of the technical specification of the CEPS system, which we realized as a μCRL specification. In this specification, all input variables are substituted by the abstract value $\mathbb{T}$. By doing so, the problem of state space explosion in the next steps of the process is avoided w.r.t. the system's interaction with environment. Afterwards, an LTS is generated from this abstracted specification. Further, this LTS is used for the actual test case generation. The generation process itself is guided by a test purpose. The scenario for the test purpose is derived from the system's functional requirements documents. From the abstract specification LTS and the test purpose a set of abstract test cases are generated. Using the original specification, these test cases are parameterized based on a CLP for actual data selection.

Here, we consider the card application `CEPCardApp` as the SUT. Doing so, the card reader application (`CEPCardReaderApp`) must be the stimulating tester. The card issuer (`CEPCardIssuerApp`) can be simulated by the card reader application since there is no direct communication between the issuer and the card (see Figure 4.5). Testing the card application thereby means stimulating it with messages and verifying the received responses whether they are plausible.

[1] see: `http://www.irisa.fr/vertecs/Equipe/Rusu/FME02/ceps.if`

For the derivation of the test purpose, we regard the use case *load transaction*. The use case *load transaction* is described by Jürjens (2005). In the following description, which is partially based on an existing NTIF specification (courtesy of the VASY team at INRIA Rhône-Alpes; cf. Garavel and Lang, 2002) of the purse card specification, all messages between the card reader application and the card are named after the NTIF card specification, while the messages between card reader application and the card issuer are named after Jürjens (2005), since there is no counterpart in the NTIF specification.

The interactions in this use case are depicted in Figure 4.6. In a first step, the reader initializes the card by sending a `CepCommand` message, parameterized with `LOADINIT`, and receives a response `CepReply`, telling that the initialization was successful (code `x9000`). Then the transaction starts. The card reader application requests money from the card issuer (`Load` message) and gets it with a response `RespL`. This money is then credited on the card by another `CepCommand` message, which is this time parameterized with `LOADCREDIT`. The response `CepReply` from the card is accepted by the card reader application and the card issuer is informed via a `Comp` message. Sets of parameters in the messages, which are irrelevant for our case, are marked with one hyphen.

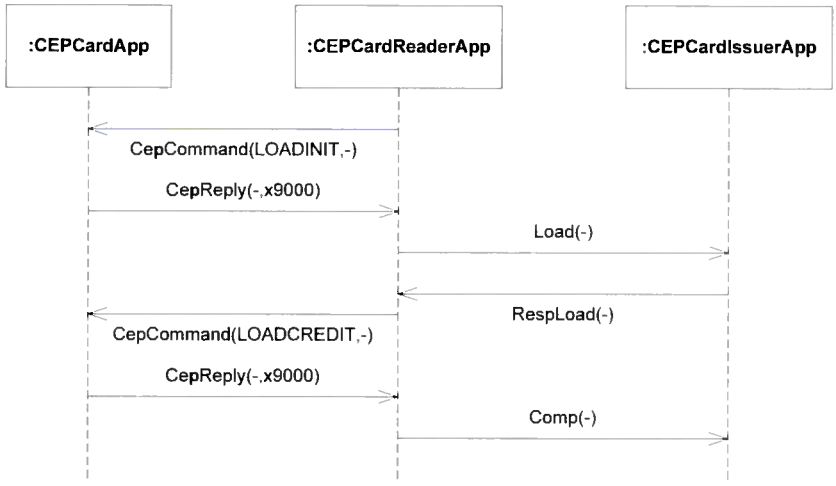

Figure 4.6: Interactions in the test purpose scenario

In the following subsections, we discuss the application of our test generation process, namely the generation of abstract test cases and the concretization and parameterization of these test cases, to the case study.

Test Case Generation

In our case, test case generation is based on the model of the SUT given as a μCRL specification. This specification is abstracted and an LTS is generated, which is then combined with the test purpose, given as an LTS, too. Depending on the states, reached in the test purpose (refusing or accepting states), verdicts are assigned to the according traces in the resulting abstract test case. Test data, and thus the possibility to identify spurious traces, is introduced by constraint-solving.

For test case generation, we first realize the NTIF specification in μCRL and simplify it. The interested reader can find some excerpts from the μCRL specification in its original and abstracted version in Appendix A. We do not want to test the logging activities of the SUT (as can already be seen in our test scenario in Figure 4.6), so that this part is not modelled. We took a part of CEPS and manually removed several interface variables, which were affected by slicing CEPS, to reduce the size of data structures. By doing so, `CommandType`, the data structure sent with the action `CepCommand`, is reduced from 15 elements to 5, `ReplyType`, the data structure for replies from the SUT, from 22 to 16 elements.

The internal variables of the SUT, which also includes internal arrays, are left untouched. These internal arrays are necessary for CEPS to, for instance, manage slots on the purse card for different currencies. For this reason, we not only have to handle arrays of data *elements*, but also arrays of data *structures*. A purse card has three slots for different reference currencies, implemented as an array with three elements. Each element is a data structure of two fields, describing one such reference currency. 16 further slots on the card refer to the reference currencies, storing amounts of money in each of these currencies. Each element of this array is again a data structure of five fields, leading to gross 80 variables for this array. A third array then stores a boolean value for each of the elements of the second array, telling whether this element has been reported or not. Due to the arrays and data structures involved, the 44 global variables for the purse card represent brutto 207 single data values, local variables not yet included. During constraint solving, which is described later in this section, the number of these internal variables that has to be handled, increases with each step of the abstract test case, ending at approx. 7,245 single data values for the smaller of the two examples (and 118,818 single values for the larger one).

In a next step, the specification is abstracted. Thereby, for every datatype, an additional abstracted datatype containing $\mathbb{T}_S$ (`TT_<datatype>`) and the lifting function κ are introduced. The abstracted specification is then parsed, and an LTS is generated. The resulting LTS is minimized modulo strong bisimulation semantics by a tool which we used for state space reduction. We experimented with two mutants of the specification. In the first mutant, a status variable of the process was after action `CepReply(updateStatus(mSlotInfo,x940A))` updated with value `x9409` instead of `x940A`. In the second mutant, this error was corrected.

In order to understand the applied state space reduction, we have to introduce the notion of *strong bisimulation*. Two automata are strongly bisimular, if they can pairwise simulate each other, whereas action labels have to really be identical (unlike

the simulation relations defined earlier in this chapter). This guarantees us that test cases generated from the reduced state space are the same like those generated from the original state space.

Definition 4.31 (Strong Bisimulation; Park, 1981). Let $\mathfrak{M}_1 = (\Sigma_1, \Lambda, \Delta_1, \sigma_{1_{init}})$ and $\mathfrak{M}_2 = (\Sigma_2, \Lambda, \Delta_2, \sigma_{2_{init}})$ be two LTSs. A binary relation $R \subseteq \Sigma_1 \times \Sigma_2$ is a strong bisimulation, if $\forall \lambda \in \Lambda . \forall \sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2 . \sigma_1 R \sigma_2$:

- $\sigma_1 \xrightarrow{\lambda} \sigma_1' \in \Delta_1 \Rightarrow \exists \sigma_2' \in \Sigma_2 : \sigma_2 \xrightarrow{\lambda} \sigma_2' \in \Delta_2 \wedge \sigma_1' R \sigma_2'$
- $\sigma_2 \xrightarrow{\lambda} \sigma_2' \in \Delta_2 \Rightarrow \exists \sigma_1' \in \Sigma_1 : \sigma_1 \xrightarrow{\lambda} \sigma_1' \in \Delta_1 \wedge \sigma_1' R \sigma_2'$

■

For the first mutant, the whole process of LTS generation and minimization took 9:26 minutes on a cluster of five 2.2GHz AMD Athlon 64 bit single CPU computers with 1 GB RAM each, running under SuSE Linux 9.3 (Blom et al., 2007). The abstracted specification had about 3.02 million states and 17.5 million transitions, which could be reduced modulo strong bisimulation to 1626 states and 5487 transitions. Finally, two single test cases without loops are generated using TGV, one of them limited to a maximal depth search for its preamble of 100 steps, the other one unlimited. Starting with the minimized abstracted system model and a test purpose of 5 states and 5 transitions, the generated unlimited test case contained 594 states with 597 transitions. The limited test case contained 108 states with 111 transitions. Test case generation took 0.65 seconds or 0.42 seconds, resp., on a workstation with one 2.2GHz AMD Athlon XP 32 bit CPU and 1 GB main memory under a Redhat Linux Fedora Core 1. For the second mutant, we received a comparable LTS in about 80 minutes, so that we can claim, that enumerative test generation with a prior application of chaotic data abstraction to CEPS can be performed in a reasonable time.

Test Case Parameterization

In this case study, we are interested in controllable test cases, preselected by TGV. Those test cases consist of only one trace to a Pass verdict and a few single steps, diverting to Inconc from this trace. For test case parameterization, we thus select the trace to Pass verdict from the generated abstract test case. Here, we are not interested in traces leading to Inconc. Parts of the trace to Pass is shown in Figure 4.7.

This test case is derived from the specification of the SUT together with the test purpose for the scenario in Figure 4.6. It is easy to recognize, that the test case contains more actions than the test purpose. The reason is, that the test purpose only sketches the main focus, while the test case must also cover initialization actions like `Power(ON)` as well as preparing actions for the test purpose action (for instance, iteration through an array to a certain position). The actions from the test case are later relevant for the determination of the test oracle.

In parallel to test case generation, a test oracle had been generated from the system's specification. The test oracle is a CLP, as it had been introduced in Chapter 3. An excerpt from of the oracle can be found in Appendix A. The test oracle is now used

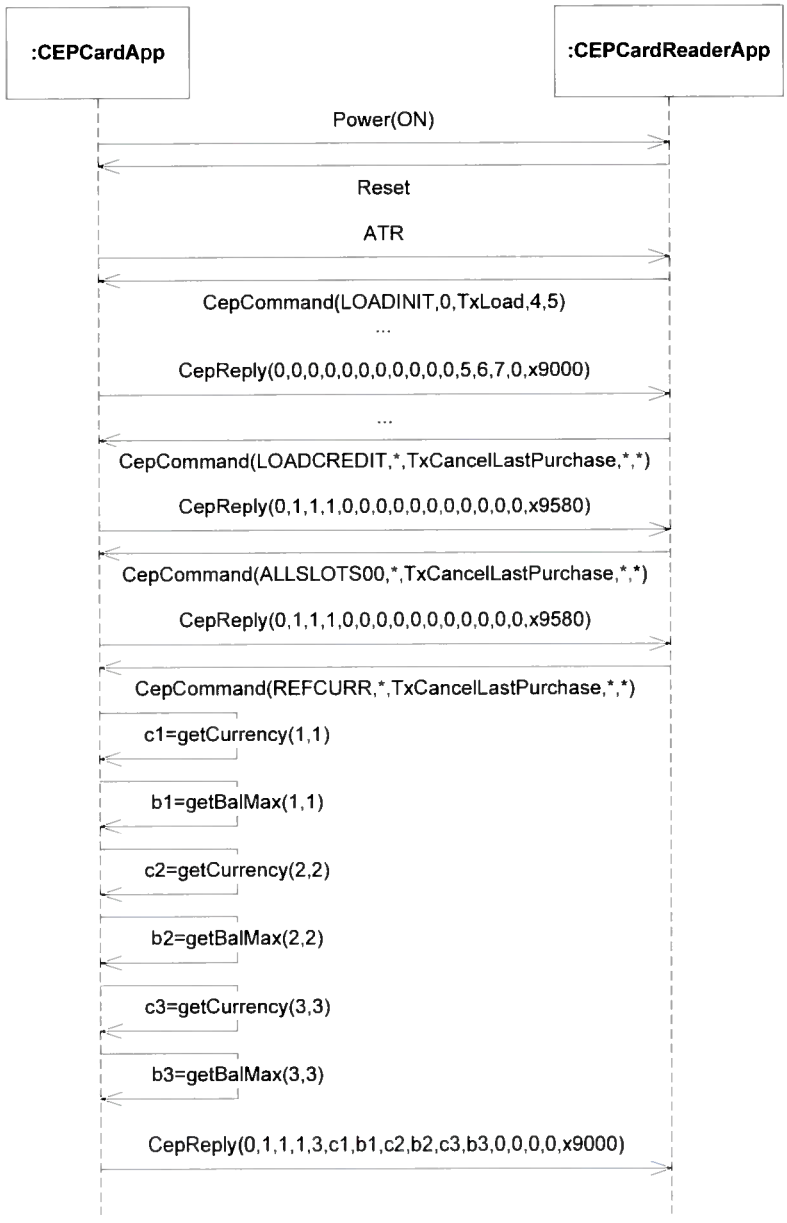

Figure 4.7: Test case example for CEPS

to find possible valuations for action parameters, with which the selected trace from the test case can be instantiated. The query to the oracle is a conjunction of a rule for the system initialization together with all rules for which there appears an action step in the abstract test case.

The first rule of the query defines the initial control location of the process as of the specification. All other parameters define initial values for the global variables. The card application has four groups of global variables, which are the *parameters of the card application* like issuer's and card identification numbers or the card's expiration date, the *static storage* of the card, its *working storage* and *messages*, which are exchanged between the card and the card reader. Since the last two groups have to be initialized, simply because they exist, but their initial values are not relevant at all, we only concentrate on some aspects of the *static storage*, the card's slots.

For our case, the card has 15 slots, of which the first 3 are used. Each of them is initialized to be in use ($\top$) and that the currency for its slot cannot be changed (also $\top$). Furthermore, each slot gets a currency identifier and a balance in this currency. In our initialization, this actual balance is equal to the maximal balance possible for this card slot. These three currencies described here are at the same time the reference currencies of this card. All the other card slots are initialized with placeholder values (0 and $\bot$). The array of reported slots, the next parameter, is completely initialized with $\bot$ since at the start time of the process, which we are describing here, no array slots can have been reported to anyone.

A limitation for Prolog CLPs and also constraint solving in general is the limitation of computer memory. Already the CEPS example leads to a vast amount of variables in our test oracle, for the limited test case 22 input and 3960 internal variables which must be introduced and which are in parts themselves data structures of, for instance, 16 elements each (e.g. `ReplyType` or the arrays; also see a previous paragraph of this section for the number of variables in the examples). At this point, the approach of constraint solving for test data selection can relatively easily reach certain limits. However, in our sample test case it is possible to calculate solutions for the constraints, so that the test case can be instantiated for execution.

Regarding the actual computation of solutions for the constraint system, again the question comes up, if a calculation in advance to the test execution or on-the-fly is the better choice. In this case, the on-the-fly solution is more promising. Calculating a complete constraint solution in advance to the test execution fixates it to exactly one trace through the SUT model. But if the SUT reacts in a nondeterministic way, this approach will lead to faulty Fail or Inconc verdicts, if the SUT chooses another trace to Pass than the one precalculated. Calculating all possible traces to Pass would also be at least very inefficient, so that an on-the-fly calculation, adapted to the actual execution trace, is the better solution here. However, test data derived from on-the-fly constraint solving can still be collected and possibly reused in later test reexecutions. The considerations of test execution and, connected to that, data selection are an integral part of the next chapter.

4.6 Related Work

Over the last years, the automation of testing has been an area of active research. In this section, we want to concentrate on those works, which are relevant for the model-based generation of test cases. The automated execution of test cases will be discussed in the next chapter.

A good overview of the area of model-based testing is given by the book of Broy et al. (2005). There, it becomes obvious, that we have to distinguish three different main strategies to testing and test generation. We have to distinguish the test of

- Finite State Machines,
- Labeled Transition Systems and
- Symbolic Transition Systems.

All these different kinds of models require different test and test generation approaches. In the remainder, we will discuss their advantages and disadvantages and will compare them to our approach. In doing so, we will concentrate on the named three approaches, since they are closest to our work. Other test generation approaches based directly on process algebras (e.g. Nogueira et al., 2007) or testing based on exploratory approaches like Directed Automated Random Testing (DART; Godefroid et al., 2005) will not be discussed here.

Finite State Machines

The approach of using Finite State Machines (FSMs) in testing must be seen a bit separately from those approaches using LTSs or STS. For this reason, we will only give a short overview along the lines of Gargantini (2005).

An FSM $\mathfrak{F} = (I, O, S, \delta, \lambda)$ is a Mealy machine defined by a *finite* set of states S, two sets of input and output symbols I and O, a state transition function $\delta : S \times I \to S$ and a state output function $\lambda : S \times I \to O$. A state $s \in S$ is left on an event $i \in I$ by a transition with target $\delta(s, i) \in S$. During the transition, an output is $\lambda(s, i) \in O$ is generated by the machine.

An FSM is tested by resetting it to its initial state and then walking along its transitions, providing inputs and checking the received outputs. Accordingly, the main objects of research are methods to reset an FSM, for instance by distinct traces, which always end in the initial state, coverage criteria and methods to meet these criteria as well as the possibility to distinguish a correct implementation of an FSM from a wrong one by only evaluating its input/output behavior.

FSM-based testing methods are often on-the-fly techniques rather than generating distinct sequences of actions (test cases), which can be applied to an FSM over and over again. For this reason, test purposes, the generation of a synchronous product of a test purpose with a specification and the explicit assignment of different types of verdicts, as we have discussed it in this chapter, are not handled by these approaches.

An advantage of FSM-based techniques is, that they already by definition do not suffer from the state space explosion problem. However, data is not handled by existing techniques, neither explicitly, which would in case of infinite domains immediately lead to an infinite state space of the Finite State Machine, nor symbolically. For this reason, the generation of parameterizable test cases is not possible with FSM-based testing techniques. An extension to FSMs, Extended Finite State Machines (EFSMs), however, also allows to consider data (cf. Cheng and Krishnakumar, 1993), leading to problems similar to the ones discussed in this chapter.

LTS-based Approaches

LTS-based approaches to testing have been studied since the papers of De Nicola and Hennessy (1983) with a later extension to input/output automata by Segala (1993) based on the theories of Lynch and Tuttle (1987). For conformance testing, LTS-based testing approaches have been extended by several conformance relations, of which we discussed the ioco relation by Tretmans (1996) earlier in this chapter.

For testing or test generation, the model of the SUT is given as a potentially infinite automaton, the LTS. As it has been defined earlier in Definition 2.7, an LTS forms a structure of states and transitions. These transitions are labeled, but those labels do not particularly distinguish action names and data parameters. In order to describe the transition of an action with variable data parameters, one must generate an own transition for any possible concrete data parameter of an action. For parameters of infinite domains, this factor leads to the well-known state space explosion problem. In most cases, this problem is surrounded by data abstraction, but that again on the cost of the data parameters.

In order to generate a test case or run a test, the given model of the SUT is examined either randomly, like, for instance, by the tool TorX (Tretmans and Brinksma, 2003) or directed by a test purpose, as this is done by TGV, which had been discussed earlier in this chapter. The latter approach logically has the advantage of generating more goal-oriented test cases. However, LTS-based test generation approaches still suffer from either state space explosion or the abstraction-induced impreciseness of data. This impreciseness covers, for instance, lost dependencies between several action parameters on one trace, hampering test data selection for testing. In our extension of TGV's test generation approach, where data dependencies are treated separately and are reintroduced for the instantiation of a test case, this problem is also solved.

The CLPs, which we generate to determine test data, are comparable to Pretschner et al. (2004a,b), where also constraint rules are generated encoding the visible inputs and outputs, guards and internal state changes. These rules are then used to generate a set of test cases by transforming a system specification in its whole into Prolog. However, in our case, test cases are already present.

STS-based Approaches

Quite close to our approach is that of *symbolic test generation* as it has extensively been studied by Rusu et al. (2000), Jéron (2004), Zinovieva-Leroux (2004) (both also in Clarke et al., 2002; Jeannet et al., 2005), and Frantzen et al. (2005). This method works directly on higher-level specifications given as IOSTSs without enumerating their state space. Given a test purpose and a specification, their product is built. The coreachability analysis is in these cases over-approximated by Abstract Interpretation (Cousot and Cousot, 1977).

The purpose and usage of abstraction techniques in our approach is conceptually different from the one of symbolic test generation, since we use a data abstraction that mitigates infinity of external data. This enables us to use existing enumerative test generation techniques for the derivation of abstract test cases which are then instantiated with concrete data derived by constraint solving. In the symbolic test generation approach, approximate coreachability analysis is used to prune paths potentially not leading to Pass-verdicts.

Both approaches, symbolic test generation as well as test generation with data abstraction, are valid for any abstraction leading to an over-approximation of the SUT's behavior. They both also employ constraint solving to choose a single test trace during test execution. However, constraint-solving is applied during test execution, which is discussed to be a performance issue (Constant et al., 2007). In our approach, a trace is preselected and constraint-solving is applied to this trace prior to the actual test run. This can, in the case of nondeterministic systems, lead to some difficulties treating the diversion of the test run from this preselected trace. This situation is handled by *Behavior-adaptation in Testing*, which is discussed in the next chapter.

Chapter 5

Behavior Adaptation in Testing

What's right is wrong
what's come has gone
what's clear and pure is not so sure
It came to me
All promises become a lie
all that's benign corrupts in time...

(Greg Graffin)

Software systems often behave nondeterministically. This nondeterminism can, for instance, be introduced by a nondeterministic specification for a deterministic system, which leaves some decisions open for its implementation phase. Such a system will later be regarded in Section 6.1. Other systems may even have a nondeterministic implementation, for instance, if they evaluate data being sent from different sources or if the system consists of several components, which act independently from each other. In this case, it is not possible to predict the order of events being sent from the system to its environment. We will regard such a system in Section 6.2.

In the previous chapter, we introduced the generation of test cases with a separate consideration of system behavior and system data. The resulting test case was a controllable one, consisting of a single trace to a Pass verdict and several single transitions from this trace to Inconc verdicts. However, for the test of a nondeterminstic system like the ones described above, such a test case is insufficient. In these cases, it is possible that during test execution the IUT leaves a trace to Pass which had been calculated beforehand and which is in principle a valid trace. When that happens, the execution of the test case has to be adapted to the new situation dynamically, otherwise, the resulting test verdicts may become useless due to a possibly large number of false positive Inconc verdicts.

In this chapter, we will develop a framework for test execution on nondeterministic IUTs, BAiT. In Section 5.1, we will develop the fundamental theory for BAiT. In Section 5.2, we will describe the realization of this framework. Afterwards, we will discuss the relation of BAiT to other test execution frameworks in Section 5.3.

5.1 A Test Execution Framework

In this section, we present the theoretical basis for test execution with behavior-adaptation. We will first present the algorithms for test execution. Then, we will link the algorithm to the ioco theory from Section 4.1 and discuss some practical aspects regarding the occurrence of quiescence in test execution.

5.1.1 Test Execution Algorithms

Here, we first give an algorithmic overview over the whole process of test selection and execution with data abstraction. Then, we describe how the activities *test selection* and *test execution* work algorithmically. Finally, we review our approach and prove the soundness of verdicts derived from test execution.

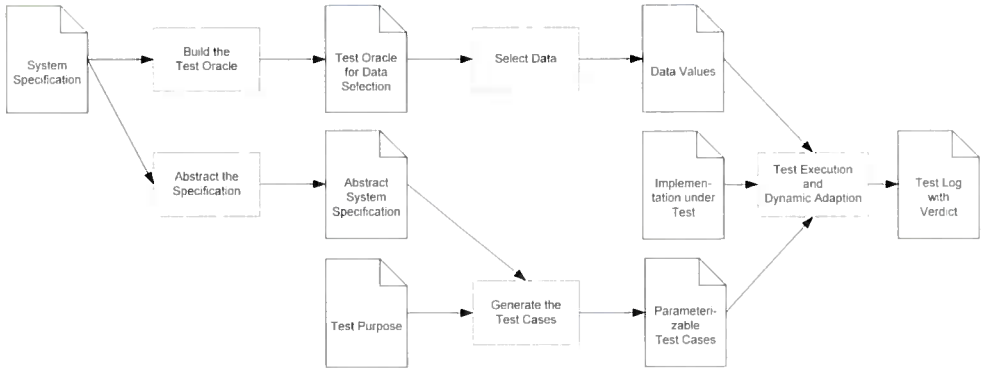

Figure 5.1: Structure of test generation and execution with BAiT

In Figure 5.1, we find an overview of the different activities for test generation and execution using BAiT. The system specification is abstracted and from the resulting control flow, parameterizable test cases are generated using TGV. In parallel, the data-interdependencies within the system are captured in a test oracle such that test data can be selected prior to or during test execution. Finally, the test is executed and results in a test log.

Algorithm 5.1 SelectAndExecTest

Require: $\mathfrak{S}, \mathfrak{M}_{\text{TP}}$
Ensure: $verdict \in \{\text{None}, \text{Pass}, \text{Inconc}, \text{Fail}\}$

```
1  setVerdict(None);
2  𝔖^𝕋 := abstract(𝔖);
3  𝔓 := buildTestOracle(𝔖);
4  𝔐^𝕋_𝔖 := generateLTS(𝔖^𝕋);
5  𝔐^𝕋_CTG := generateCTG(𝔐^𝕋_𝔖, 𝔐_TP);
6  (π, θ) := NewPassTrace(no_trace, ∅, 𝔐^𝕋_CTG);
7  if (π, θ) ≠ no_solution then
8     ExecTest(π, θ, no_trace, 𝔐^𝕋_TC);
9     terminate;
10 fi
```

Algorithm 5.1 describes the test process in more detail. Its input parameters are a specification $\mathfrak{S}$ and a test purpose $\mathfrak{M}_{TP}$. $\mathfrak{S}$ is abstracted to $\mathfrak{S}^{\mathbb{T}}$ according to Table 4.1. Then $\mathfrak{M}^{\mathbb{T}}_{\mathfrak{S}}$ is generated from $\mathfrak{S}^{\mathbb{T}}$. In parallel, a test oracle $\mathfrak{P}$ is built according to Chapter 3, containing all conditions from $\mathfrak{S}$. $\mathfrak{P}$ will later be needed to select test data during test execution. From the two IOLTSs, the complete test graph $\mathfrak{M}^{\mathbb{T}}_{CTG}$ is generated using TGV. $\mathfrak{M}^{\mathbb{T}}_{CTG}$ may contain choices between several outputs to the IUT or even between inputs and outputs, so it is not necessarily controllable. Furthermore, $\mathfrak{M}^{\mathbb{T}}_{CTG}$ is an overapproximation of all test cases of the original system which satisfy the test purpose, so it may contain traces leading to unsound verdicts. The tester together with the test oracle assure during test execution, that these *potentially* unsound verdicts do not turn into *actual* unsound verdicts. We will prove this later in this section.

The algorithm NewPassTrace plays a crucial role in trace selection for test execution. It is not only refered to by the algorithm SelectAndExecTest, but also by ExecTest, which actually executes a particular test trace.

Algorithm 5.2 NewPassTrace

Require: $\beta, \theta, \mathfrak{M}^{\mathbb{T}}_{CTG}$
Ensure: $(\pi, \theta') \in [\![\mathfrak{M}^{\mathbb{T}}_{CTG}]\!]_{Pass} \times \{Var \to \mathfrak{D}\}$

```
1  π := selectFirst(β, [[M^T_CTG]]_Pass);
2  while π ≠ no_trace do
3     O_π := oracle(0, π);
4     θ' := solve(P, O_π, ⌊θ⌋_β);
5     q := O_π(θ');
6     if ⟨q, ⊤⟩ is satisfiable then
7        return (π, θ');
8     else
9        π := selectNext(β, [[M^T_CTG]]_Pass);
10    fi
11 od
12 return (no_solution);
```

NewPassTrace is shown in Algorithm 5.2. The algorithm takes a trace prefix β, a valuation θ and a test case $\mathfrak{M}^{\mathbb{T}}_{CTG}$ as input parameters and returns a trace $\pi \in [\![\mathfrak{M}^{\mathbb{T}}_{CTG}]\!]_{Pass}$ as well as an appropriate valuation (here θ'). It iterates over all possible traces to Pass with prefix β in the test case and returns the first, which contains β as its prefix and satisfies the query q under the valuation θ'. θ' is derived by solving the CLP $\mathfrak{P}$ for the trace π with a partial solution $\lfloor\theta\rfloor_\beta$ given. This partial solution cannot be changed anymore, since it gives the (proper) valuation for the already executed trace β. The new trace found by NewPassTrace must satisfy $\langle \mathfrak{O}_\pi(\theta'), \top \rangle$. If it does not, then the next possible trace to Pass is selected.

Let π be a Pass-trace of $\mathfrak{M}^{\mathbb{T}}_{CTG}$, θ be a solution for the query obtained from π by the rules in Figure 4.4, and β be the already executed prefix of π, which is initially empty. Let next be a function that returns the next step of trace π or no_step, if no such step

Algorithm 5.3 ExecTest

Require: $\pi, \theta, \beta, \mathfrak{M}^{\mathbb{T}}_{CTG}$
Ensure: $verdict \in \{None, Pass, Inconc, Fail\}$

```
1  step := next(π, β);
2  if step = no_step then
3    if |β| > 0 then
4      setVerdict(Pass);
5    else
6      setVerdict(None);
7    fi
8  else if step = τ then
9    ExecTest(π, θ, add(β, step), 𝔐^𝕋_CTG);
10 else if step =!s(x) then
11   sendToIUT(s(⟦x⟧_θ));
12   ExecTest(π, θ, add(β, step), 𝔐^𝕋_CTG);
13 else if step =?s(x) then
14   receiveFromIUT(sig(y));
15   if sig = s ∧ ⟦y⟧ = ⟦x⟧_θ then
16     ExecTest(π, θ, add(β, step), 𝔐^𝕋_CTG);
17   else
18     β' := add(β, sig(y));
19     𝔒_β' := oracle(0, β');
20     q_β' := 𝔒_β'(⌊θ⌋_β[x↦⟦y⟧]);
21     if ¬satisfiable(⟨q_β', ⊤⟩) then
22       setVerdict(Fail);
23     else
24       (π', θ') := NewPassTrace(β', ⌊θ⌋_β[x↦⟦y⟧], 𝔐^𝕋_CTG);
25       if (π', θ') = no_solution then
26         setVerdict(Inconc);
27       else
28         ExecTest(π', θ', β', 𝔐^𝕋_CTG);
29       fi
30     fi
31   fi
32 fi
```

exists. Sending a signal to the IUT happens by the function sendToIUT, receiving by receiveFromIUT. Both functions are parameterized with the sent or received signal.

Test execution works as described in the recursive algorithm in Algorithm 5.3. First, the actual step under consideration is computed. Then a decision is made, based on the type of this step. If the next step is no_step, that means no further step has been found in the test trace. Then the algorithm assigns either the None verdict, if no steps have yet been executed, or the Pass verdict. In this case, the test execution finished without finding any failures or inconclusive situations in the IUT. If the next step is a τ-step, ExecTest is invoked recursively, adding the τ-step to the trace prefix, which has already been executed. An output step $!s(x)$ is treated nearly equally, except that s is sent to the IUT. Its parameters are instantiated according to θ.

Treating Internal Steps in Blackbox Testing

This algorithm is designed in a way, that τ-steps are de facto part of the communication between the tester and the IUT. In practice, this is not the case, since τ-steps are internal steps in the IUT. In the practical realization of the algorithms of this section, which is described in Section 5.2, τ-steps will be "executed" by assuming that they appear in an executed trace.

Of course, this assumption does not have to be true. If there is logic in the SUT determining the amount of τ-steps to be executed in the IUT, BAiT can get out of sync with the IUT. E.g. if two traces $a.\tau.\tau.b$ and $a.\tau.b$ are defined in the specification of the IUT and the IUT executes the trace $a.\tau.\tau.b$ while BAiT assumes $a.\tau.b$ has been executed, we get into such a situation. The yet executed trace stub is only extended during a test run, but never substantially changed in its structure. This means, that BAiT has no possibility to adapt this part in order to correct its wrong assumption about the progress of the test run. For this reason, such a test run will – in practice – lead to an unsound verdict.

However, the question is still open, why to treat *internal* steps of a system in a *blackbox* test anyway. The reason is, that some actions, like assignments, can be encoded in this way, but it is important to assure, that there is no logic defined in the SUT, which makes it possible to execute different amounts of (invisible) internal steps between two (visible) actions. If this is assured, BAiT may also safely assume internal τ-steps in a blackbox test.

Handling an input $?s(x)$ is more complex. First, the input is received from the IUT as $?sig(y)$. If now both the signal sig and the valuation of its parameters $[\![y]\!]$ are as expected, then the step is just added to the executed trace prefix and a recursive invocation of the execution algorithm happens. If the signal sig or the parameter valuation does not fit the expectations, then it is checked, whether test execution has already left the valid traces in the system specification. In this case, Fail is assigned, otherwise a new trace to Pass with the new valuation is searched. If no such trace exists, Inconc is assigned. Otherwise, the algorithm is invoked recursively and test execution goes on.

We argue the correctness of our approach by proving that the verdicts assigned to the IUT after having applied the algorithm ExecTest, are sound. From the generated

CTG $\mathfrak{M}^{\mathbb{T}}_{CTG}$, we select a trace π to Pass. This trace is instantiated with data, which has been derived from a query to $\mathfrak{P}$. The trace π is then executed. An already executed prefix of π is trace β.

In the following, we will first prove some invariants over the algorithm. Then we prove that the test verdict assigned to a test case after execution of the algorithm is sound. All line numbers in the proofs refer to those in ExecTest (Algorithm 5.3).

Lemma 5.1. Given a finite trace π, the test execution algorithm always terminates, given that the IUT has no deadlocks or livelocks. ■

Proof. As its first statement, the algorithm ExecTest always executes a function next on trace π to derive the next step in the test to execute. Given a finite trace π, at the end of this trace the function returns no_step. In this case, the trace has been executed completely and the algorithm terminates with either verdict Pass or None (lines 4 and 6).

The function next determines the next step in π by comparing it to its already executed prefix β. Each step executed during the test is appended to β before the test execution algorithm is reinvoked. This happens in lines 9, 12, 16 and in line 18 before the reinvocation of ExecTest (line 28). In so doing, it is guaranteed that β always reflects the actual state of test execution and that next always returns a correct next step or no_step after π has been executed completely. □

Lemma 5.2. When the test execution algorithm terminates, it always assigns a verdict. ■

Proof. The test execution algorithm either completely executes π and then terminates assigning Pass or None (see Lemma 5.1), or it already terminates in line 26 assigning Inconc or in line 22 assigning Fail. In all other cases (lines 9, 12, 16 and 28) it is reinvoked and does not terminate with the actual step. □

Lemma 5.3. For all $\beta(\theta)$, for which the algorithm does not terminate with a Fail verdict, holds: $\beta(\theta) \in [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{traces}$. ■

Proof. This lemma is proven by induction over the length of trace π.

First step: The initial trace π has been chosen by $NewPassTrace(no\_trace, \emptyset, \mathfrak{M}^{\mathbb{T}}_{CTG})$ for execution. In the invocation of this algorithm, no trace prefix is preselected (parameter no_trace) and the possible resulting valuation for the chosen trace has also not been limited (parameter $\emptyset$). The selection made by the Pass trace selection algorithm is the trace π which is per definition (Definition 4.14) a trace in $[\![\mathfrak{M}^{\mathbb{T}}_{\mathfrak{S}}]\!]_{traces}$. The trace can be instantiated by θ to $\pi(\theta) \in [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{traces}$, since NewPassTrace ensures $\mathfrak{O}_\pi := oracle(0, \pi) \wedge \theta := solve(\mathfrak{P}, \mathfrak{O}_\pi, \mathfrak{M}^{\mathbb{T}}_{CTG}) \wedge q := \mathfrak{O}_\pi(\theta) \wedge \langle q, \top \rangle$ is satisfiable, which holds if and only if $\pi(\theta) \in [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{traces}$ (Lemma 4.30 and proof). Since $\beta(\theta)$ is a – possibly empty – prefix of $\pi(\theta)$, the aforementioned claim also holds for $\beta(\theta)$.

Inductive step: The test execution algorithms recursively executes $\pi(\theta)$ transition by transition with $\beta(\theta)$ being the prefix, which has already been executed. Taking such an arbitrary transition, we now have to regard the recursive invocation of ExecTest, whether $\pi'(\theta') \in [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{\text{traces}}$ still holds if $\pi(\theta) \in [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{\text{traces}}$.

1. First we have to regard the recursive invocation in lines 9, 12 and 16. In all three cases $\pi' = \pi \wedge \theta' = \theta$ so that our claim holds (trivial case). The already executed prefix of π' is $\beta'(\theta')$ in this case, for which of course $\beta'(\theta') \in [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{\text{traces}}$ also holds, since neither π nor θ have changed.
2. In line 28, both $\pi' \neq \pi \wedge \theta' \neq \theta$. In this case the new trace π' to Pass is searched by NewPassTrace, and executed only if $\mathfrak{O}_{\pi'} := \mathtt{oracle}(0, \pi') \wedge \theta' := \mathtt{solve}(\mathfrak{P}, \mathfrak{O}_{\pi'}, \mathfrak{M}^{\mathbb{I}}_{\text{CTG}}) \wedge q := \mathfrak{O}_{\pi'}(\theta') \wedge \langle q, \top \rangle$ is satisfiable, which holds if and only if $\pi'(\theta') \in [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{\text{traces}}$. Thus for the already executed prefix of $\pi'(\theta')$ also holds: $\beta'(\theta') \in [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{\text{traces}}$.

In all cases, where $\pi \notin [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{\text{traces}}$, this is discovered and test execution terminates with a Fail verdict (line 22 and appropriate proof).

□

Lemma 5.4. In case, that the verdict Fail is assigned, for the trace $\beta' = \mathtt{add}(\beta, \mathtt{sig}(y))$ holds: $\beta'(\theta_{[x \mapsto [\![y]\!]]}) \notin [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{\text{traces}}$. ■

Proof. First of all, the Fail verdict is assigned only in line 22, where input from the IUT is evaluated. It is checked whether the executed trace $\beta' = \mathtt{add}(\beta, \mathtt{sig}(y)) \in [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{\text{traces}}$ under $\lfloor\theta\rfloor_{\beta[x \mapsto [\![y]\!]]}$.

The valuation θ has been precalculated for the whole test sequence π. $\lfloor\theta\rfloor_{\beta}$ denotes that part of θ, which is relevant for the subtrace β. Accordingly, $\lfloor\theta\rfloor_{\beta[x \mapsto [\![y]\!]]}$ denotes the same part of the valuation with x being set to the value of y.

A trace is valid only if $q_{\beta'} = \mathfrak{O}_{\beta'}(\lfloor\theta\rfloor_{\beta[x \mapsto y]}) \wedge \langle q_{\beta'}, \top \rangle$ is satisfiable (see Lemma 4.30 and proof). The verdict Fail is set only in those cases where $\langle q_{\beta'}, \top \rangle$ is not satisfiable and thus $\beta'(\lfloor\theta\rfloor_{\beta[x \mapsto [\![y]\!]]}) \notin [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{\text{traces}}$. For this reason, the assignment of the Fail verdict is sound. □

Lemma 5.5. In case, that the verdict Inconc is assigned, for the executed trace β holds: $\beta \in [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{\text{traces}} \wedge \beta \notin [\![\mathfrak{M}_{\text{CTG}}]\!]_{\text{Pass}}$. ■

Proof. The verdict Inconc is assigned in line 26.

In this case $\beta'(\theta_{[x \mapsto [\![y]\!]]})$, consisting of the previously executed trace β and the action under consideration $\mathtt{sig}(y)$ (both under valuation $\theta_{[x \mapsto [\![y]\!]]}$) is a trace from $[\![\mathfrak{M}_{\mathfrak{S}}]\!]_{\text{traces}}$ ($q := \mathfrak{O}_{\beta'}(\lfloor\theta\rfloor_{\beta[x \mapsto [\![y]\!]]}) \wedge \langle q, \top \rangle$ is satisfiable; cf. Lemma 4.30), and no further trace to a Pass verdict could be found, so that NewPassTrace returns no_solution. This means, that either no trace has been found in the test case, or a trace π' has been found, but $\pi' = \mathtt{NewPassTrace}(\beta', \lfloor\theta\rfloor_{\beta[x \mapsto [\![y]\!]]}, \mathfrak{M}^{\mathbb{I}}_{\text{CTG}}) \wedge \mathfrak{O}_{\pi'} = \mathtt{oracle}(0, \pi') \wedge \theta' = \mathtt{solve}(\mathfrak{P}, \mathfrak{O}_{\pi'}, \lfloor\theta\rfloor_{\beta[x \mapsto [\![y]\!]]}) \wedge q := \mathfrak{O}_{\pi'}(\theta') \wedge \langle q, \top \rangle$ is not satisfiable. In both cases, verdict Inconc has to be assigned per definition (see Definition 4.14). Thus, the assignments of verdict Inconc in the algorithm ExecTest are sound. □

Lemma 5.6. In case, that the verdict Pass is assigned, the executed trace $\beta(\theta) \in [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{traces} \wedge \beta \in [\![\mathfrak{M}^{\mathbb{T}}_{CTG}]\!]_{Pass} \wedge |\pi| > 0$. In case that $|\pi| = 0$, None is assigned. ■

Proof. The Pass verdict is assigned in line 4 in those cases only, where a trace $\pi(\theta) \in [\![\mathfrak{M}_{\mathfrak{S}}]\!]_{traces}$ (cf. Lemma 5.3), with π having been found in $\mathfrak{M}^{\mathbb{T}}_{CTG}$ by NewPassTrace, could be executed to its very end without any Fail or Inconc verdicts assigned. Under these conditions and if the executed trace has at least one transition, then assigning the Pass verdict is sound (see line 4). In those cases, where no transition had been executed, setting the None verdict is sound (see line 6). □

Proposition 5.7. The assignment of the test verdict to a test trace is sound. ■

Proof. This proposition immediately results from the three previous lemmata. □

5.1.2 Test Execution Regarding ioco

Up to this point, Algorithm 5.3 is correct w.r.t. ioconf, as it has been introduced in Section 4.1 (Definition 4.2). However, ioco is not covered, since the algorithm does not interpret an input δ from the IUT. In order to also consider quiescence, we have to modify the algorithm from line 13 on.

We extended the original algorithm in Algorithm 5.4 by the lines 8 to 12. This modification allows to silently accept possibly appearing δ-steps announcing quiescence. If a δ-step occurs that is not allowed in the specification, this δ-step is also not possible in the test case, since it does not appear in the synchronous product $\mathfrak{M}^{\mathbb{T}} \times \mathfrak{M}_{TP}$. In this case, appearance of δ violates the conformance of IUT with $\mathfrak{M}$ and thus, the verdict is set to Fail and test execution terminates. If it is allowed, however, its occurrence is ignored and test execution proceeds. The drawback is, that if δ is allowed in the specification and it is specified as a δ-loop, test execution not necessarily terminates.

In practice this approach also has the drawback, that most IUTs do not announce quiescence by sending a δ to the tester. User interfaces waiting for user input showing a prompt are an exception for this. In systems without this feature, δ can only be assumed, if we are waiting for input from the IUT. In case, the tester has to provide output, i.e. in a step $\delta.!s(x)$ (the standard case for quiescence), it will provide this output. In a step $\delta.\tau$, the tester will first "execute" (i.e. ignore) the τ-step and postpone the δ. This happens, because the tester is – due to the blackbox-setting of our test – not aware of internal steps happening in the IUT (δ can be possible in a test case, but will not be actively selected by the test trace selection algorithm). So the only possibility to really observe δ is an input from the IUT: $\delta.?s(x)$. The δ will be observable by the occurrence of a timeout. There are three possible solutions w.r.t. handling this timeout:

Assign Fail: The system does not conform, since it should not produce an output δ. This is the theoretical solution, which we have chosen in this section. However, we cannot be sure, that the system will produce a valid output a bit later (cf.

also Zinovieva-Leroux, 2004). We would base this verdict on some kind of *virtual* output, which is actually not even produced by the IUT.

Assign Inconc: The logical consequence of the above would be to assign an Inconc verdict instead. However, this verdict could be premature (see below).

Search another trace: The system might, however, just be waiting for another input. In this case, it would be perfectly ioco for so far, so that assigning any verdict does not make sense anyway here. Testing could then go on, instead of prematurely assigning a verdict. This is the variant, practically implemented at the moment.

Algorithm 5.4 ExecTest′

Require: $\pi, \theta, \beta, \mathfrak{M}^{\mathbb{T}}_{CTG}$
Ensure: $verdict \in \{None, Pass, Inconc, Fail\}$

```
1  if step = no_step then
2    ...                                   // see first three cases of Algorithm 5.3
3  else if step =?s(x) then
4    receiveFromIUT(sig(y));
5    if sig = s ∧ [[y]] = [[x]]_θ then
6      ExecTest′(π, θ, add(β, step), 𝔐^𝕋_CTG);
7    else
8      if sig = δ then
9        if δ ∉ in(𝔐^𝕋_CTG after β) then
10         setVerdict(Fail);
11       fi
12     else
13       β′ := add(β, sig(y));
14       𝔒_β′ := oracle(0, β′);
15       q_β′ := 𝔒_β′(⌊θ⌋_β[x↦[[y]]]);
16       if ¬satisfiable(⟨q_β′, ⊤⟩) then
17         setVerdict(Fail);
18       else
19         (π′, θ′) := NewPassTrace(β′, ⌊θ⌋_β[x↦[[y]]], 𝔐^𝕋_CTG);
20         if (π′, θ′) = no_solution then
21           setVerdict(Inconc);
22         else
23           ExecTest′(π′, θ′, β′, 𝔐^𝕋_CTG);
24         fi
25       fi
26     fi
27   fi
28 fi
```

This implemented variant of assigning verdicts to tests has the advantage, that it avoids Inconc verdicts as much as possible. It does, however, have the disadvan-

tage, that searching an alternative trace might never end. This is, for instance, the case if there is a loop in the CTG which BAiT unsuccessfully tries to unfold without ever being able to reach a Pass verdict from the current state of the test run and the IUT. In order to prevent such livelocks in the tester, we have introduced a threshold in the tool. This threshold can be configured for a test run and defines the maximum number of tries to find a trace to Pass before giving up. By that, infinite loops within the tester are avoided, however, setting this threshold too low reintroduces the problem of false positive Inconc verdicts. Configuring the threshold optimally thus needs some expertise.

Another issue not yet concerned are timeouts. An IUT might not react for a short moment, but might proceed further properly after that time. If the tester does not take such timing behavior into account, it might already be searching for and executing an alternative test trace and by that violate the test results. In order to avoid this problem, one can use a timed specification language. Such a language gives the possibility to test the IUT a timed ioco conformance. Since we are using untimed μCRL specifications, we solve this problem with a global timeout threshold, which can be configured on a per-testrun setting as the trace search threshold can. Only if the system does not react within this timing setting, BAiT will try to find an alternative test trace.

At this point, we cannot come to a conclusive decision about the verdict to assign. In the remainder of this section, we will discuss timed alternatives for ioco instead of the standard untimed one in order to decide for proper test verdicts. Finally, we will outline a simplified approach, which we follow in this thesis.

Timed ioco

To overcome the problems, which we have just described, the relations tioco (also named $\sqsubseteq_{\text{tioco}}$; Brandán Briones and Brinksma, 2005), tioco (sic!; Krichen and Tripakis, 2004, 2006) and rtioco (Larsen et al., 2005) have been developed. The latter two conformance relations are proven equivalent by Krichen and Tripakis (2006), while the first relation is different despite its equal name. Instead of an untimed quiescence, both tioco by Krichen and Tripakis and rtioco are defined on timed traces, i.e. on traces whose transitions are not only labeled with actions $\lambda \in \Lambda$, but also with delays $t \in \mathbb{R}_{\geqslant 0}$.

We will now exemplarily describe the relation tioco by Krichen and Tripakis to give the reader an insight into the material. In the course of the discussion, we will consider only relative time, i.e. delays of actions w.r.t. transitions in a system. We will, however, not combine these theoretical results into our work on test execution with BAiT, since any timed variant of ioco requires a timed specification language to be used. We are using untimed μCRL in this thesis and we will at the end of this section introduce our practical approach for test execution using a global timer.

Definition 5.8 (Timed Input/Output Labeled Transition System (TIOLTS); Krichen and Tripakis, 2006). A TIOLTS is a transition system $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{\text{init}})$ with $\Lambda =$

$\Lambda_{in} \cup \Lambda_{out} \cup \{\tau\}$ the set of action labels and $\Delta \subseteq \Sigma \times (\Lambda \cup \mathbb{R}_{\geqslant 0}) \times \Sigma$ the transition relation. The set of transitions is divided into discrete Δ_d and timed Δ_t, s.t. $\Delta = \Delta_d \cup \Delta_t$. A transition can be either labeled with an action $\lambda \in \Lambda$ ($\sigma \xrightarrow{\lambda} \hat{\sigma} \in \Delta_d$) or a delay $t \in \mathbb{R}_{\geqslant 0}$ ($\sigma \xrightarrow{t} \hat{\sigma} \in \Delta_t$). ■

All further definitions in Krichen and Tripakis (2006), however, are defined on Timed Automata with Inputs and Outputs (TAIOs).

Definition 5.9 (Timed Automaton with Inputs and Outputs; ibid). A TAIO is an automaton $\mathfrak{S} = (L, C, A, E, \ell_{init})$ with $\Lambda = \Lambda_{in} \cup \Lambda_{out}$ (no τ if observable) the set of action labels and C a set of clocks. A state in a TAIO is a pair $\sigma = (\ell, c_t) \in L \times C$. An edge is a tuple $(\ell, \hat{\ell}, g, r, d, \iota)$ with ℓ, $\hat{\ell}$ the source and destination locations of the edge, $\iota \in A$ an action, g a guard regarding the timers in the system, $r \in C$ the set of timers to be resetted to 0 and d a *deadline*, which states whether the action is delayable or not. ■

Definition 5.10 (Function out; ibid). The function $out : 2^{L \times C} \rightarrow 2^{A \cup \mathbb{R}_{>0}}$ is defined for a set of states $\Sigma' \subseteq L \times C$ as:

$$out(\Sigma') = \bigcup_{\sigma \in \Sigma'} (\{a | a \in A_{out} \wedge \exists \sigma \in \Sigma' \exists \hat{\sigma} \in \Sigma : \sigma \xrightarrow{\lambda} \hat{\sigma}\} \cup \{t | t \in \mathbb{R}_{>0} \wedge \exists \sigma \in \Sigma' \exists \hat{\sigma} \in \Sigma : \sigma \xrightarrow{t} \hat{\sigma}\})$$

■

The set $OTT(\mathfrak{M})$ defines the *observable timed finite-length real-time traces* of $\mathfrak{M}$. Based on these traces, the conformance relation tioco can be defined as follows for a specification $\mathfrak{M}$ and an implementation $\mathfrak{I}$:

Definition 5.11 (tioco; ibid).

$$\mathfrak{I} \text{ tioco } \mathfrak{M} \Leftrightarrow \forall \pi \in OTT(\mathfrak{M}) : out(\mathfrak{I} \text{ after } \pi) \subseteq out(\mathfrak{M} \text{ after } \pi)$$

■

Defining a Timed System for Test Generation and Execution

Timed system specifications, from which test cases can be generated, can in principle be defined using Timed μCRL (Groote, 1997). For Timed μCRL there exists a `delay` predicate (ibid, Section D.1), which can be used for transitions $\sigma \xrightarrow{t} \hat{\sigma} \in \Delta_t$ in TIOLTSs. While, in an TIOLTS a timed transition has the meaning of an exact delay, while delays in Timed μCRL have an *at least* semantics. This means, that the system can actually delay longer than the specified delay. In an TAIO, this issue is approached by the *deadline* parameter of an edge, which allows to specify, whether a timer might or might not be exceeded.

The synchronous product of two automata $\mathfrak{M}_t \times \mathfrak{M}_{TP}$ with $\mathfrak{M}_t$ being derived from a specification in Timed μCRL should result in a timed test case, in which occurrences of `delay` can be interpreted by the test executor as timeout settings. If the timeout is

not met by the IUT, test cases then would clearly be assigned Fail. This also, because a time delay would only occur, if we are waiting for input from the IUT as defined by the specification. However, at the moment of writing this thesis, it is not possible to generate a state space from a specification in Timed μCRL with the existing tools.

As a solution to this problem, several extensions to the untimed variant of μCRL have been proposed. Blom et al. (2003) introduce a distinct action tick in order to explicate discrete time steps, an approach which has been further extended to μCRL^{tick} by Wijs (2007), also regarding the system-internal communication behavior under discrete timing. For the practical purpose of computing an allowed delay and timing out during test, these approaches are only of limited use. Since every single possible time progress step has to be defined separately, the approach does not scale with the resolution of discrete time events. For instance, a time delay of 1 second expressed in a discrete time resolution of milliseconds needs 1,000 tick-steps. Furthermore, such a delay does not only have to be specified, but also has to be evaluated. For that, one needs to follow the trace and count the tick-steps. This also shows, that the approach is suboptimal for our purposes.

In the remainder of this thesis, we will not consider system specifications with explicit timing information. In order to realize a somewhat timed version of ioco for practical test execution, the test executor of BAiT, as it will be realized in the following section, is based on a general timeout. If this timeout is exceeded, BAiT considers having received quiescence. Since there is no information given in the system specification, whether quiescence is allowed at that point of execution, BAiT will not directly assign any verdict, but it will try to adapt test execution. If this is not possible, then BAiT will assign the test verdict Inconc to the yet executed part of the test trace.

5.2 Realization of the Test Execution Framework

In this section, we will discuss the technicalities in the practical realization of the test execution framework. We will first present our main goals and decisions on a high level. Then, we will discuss the general architecture of the tool BAiT, followed by a more detailed description of its components regarding test data selection and management, test trace selection and test execution.

5.2.1 Goals in the Development of BAiT

The goal for the development of BAiT was the implementation of the test execution algorithm, which has been presented earlier in this chapter. This algorithm could not be implemented in a straight-forward way, since several technical issues had to be considered. In the theoretical algorithm, exchanging action events with the IUT was formulated at a high level of abstraction. Also providing queries to the test oracle and solving them using a constraint solver, were formulated abstractly, and so was trace selection from a *set of traces*. In the realization of the algorithm, we

had to first decide for a concrete system platform for the tool and the possible IUTs. Furthermore, we had to technically solve the integration of the test generator TGV and a constraint solver. Last, we had solve the integration of algorithms for test trace and test data selection into BAiT.

For the system platform we decided for Java-based systems. Java programs are executed based on a virtual machine. This means, that these programs support sophisticated features like the dynamic selection and invocation of methods (reflection API), which we will make use of in the development of BAiT. As a further decision, tester and IUT communicate with each other in a method-based setting, i.e. messages are exchanged by mutual method invocations. Since our specification language μCRL does not support actions with return types, we decided to implement a bidirectional communication between the tester and the IUT, which is based on the *Observer Pattern* proposed by Gamma et al. (1995).

In order to perform data selection, we had to integrate a constraint solver. We decided to use ECLiPSe Prolog, and more particularly its libraries *ic* and *ic_symbolic* (Brisset et al., 2006) for numerical and enumerative (symbolic) data domains. Hardly any code directly affected by this choice is, however, implemented in the test execution framework, but in the generated test oracle itself. This issue has already been discussed in Chapter 3.

The second tool to integrate was the test case generator TGV. It generates test cases as CTGs, which must be parsed and examined for traces to respectively `Pass` and `Inconc` verdicts. While examination of LTSs is part of the test execution algorithm, technicalities regarding the data format of TGV have been implemented in a separate library, which we do not regard here.

The realization of framework BAiT followed two main lines. The first one was separation of concerns (*divide et impera*). It was important to avoid mixing functional code, i.e. the algorithms `NewPassTrace` and `ExecTest` themselves, and non-functional code, i.e. the interaction with basic libraries and the management of data structures, as much as possible to improve the testability of the framework itself and to ease debugging. The second line was a focus on *extensibility*. This means, that certain parts of the algorithms, for instance trace selection and even more data selection, can be adapted to a particular test requirement (like coverage criteria and the like) by providing custom algorithms for these aspects and plugging them into the core of BAiT. We will discuss this issue in more detail later in this section. Extensibility was reached by consequently *façading* all packages with interfaces (Gamma et al., 1995).

In the remainder of this section, we will now give an overview of the architecture of BAiT and will then discuss the design decisions made for its single components. We will also, in place, provide a technical discussion against the runtime environment of TTCN-3. A more general discussion of the differences between BAiT and TTCN-3 will follow in Section 5.3.

5.2.2 Test Generation and Test Execution with BAiT

In order to generate and execute a test, several artifacts have to be provided:

System Specification: The system specification defines both data- and control-related aspects, which are implemented in the IUT. It is the starting point for the data-centered test generation and – together with the test purpose – also for the control-centered generation.

Test Purpose: The test purpose is the specification of a test scenario. It guides the control-centered test generation by sketching out relevant actions in a particular order without having to be complete (i.e., naming all actions of a particular trace).

Proxy Classes: Proxy classes serve as the platform- and system-specific connector between the generated test cases and the actual IUT.

IUT: The IUT, finally, is the implementation of the software, which is tested.

Test cases are generated from the system specification and the test purpose as has been discussed in the previous chapter. The test oracle, necessary to parameterize a test case, is also generated from the system specification. This issue has mainly been discussed in Chapter 3. Finally, in order to set up a test for execution and to communicate with the IUT, we need proxy classes. Those can mainly be automatically generated from the interfaces of the IUT, but can also be manually extended for the purpose of customization. A reason to extend these classes would, for instance, be the setup of a special manner of test logging.

5.2.3 General Architecture of BAiT

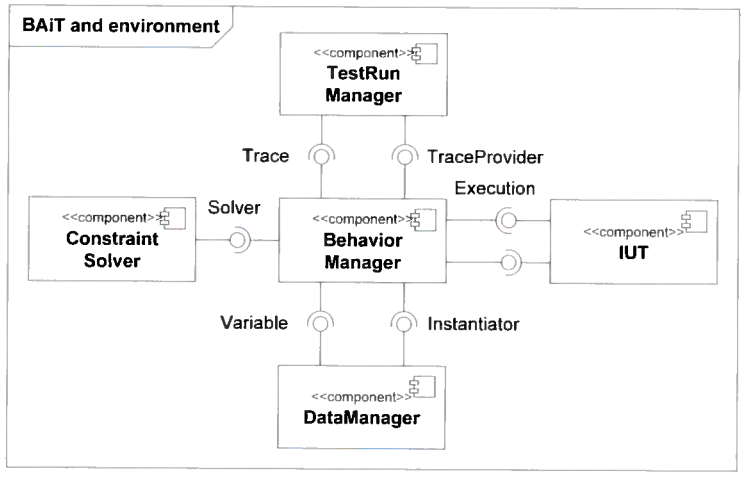

Figure 5.2: Component structure of BAiT

In Figure 5.2, we can see the structure of components of BAiT. Two of the components shown in the diagram form the environment of the tool. Those are on the one hand the IUT and on the other hand the constraint solver (component `ConstraintSolver`). The three components of the test executor itself are the `TestRunManager`, the `BehaviorManager` and the `DataManager`. When a test is executed, the `TestRunManager` creates an instance of the IUT if necessary and then connects to it. After this has successfully happened, a possible trace is computed for the test run. In order to do so, the `BehaviorManager` takes over and searches for an applicable trace in the test cases. Such a trace is a sequence of actions (steps, which are also part of the behavior and thus of component `BehaviorManager`) with parameters. Parameters are data elements, whose management is implemented in the component `DataManager`.

Then, each such trace is transformed into a query to the constraint solver. The behavior manager queries the constraint solver and retrieves a term with the solution of the query. If there is no such solution, further steps have to be taken by the behavior manager in order to find a test trace. If a solution has been found, the term received from the constraint solver amongst other things contains ranges for action parameters or already fully instantiated parameters. This information is provided to the data manager, which subsequently tries to instantiate those parameters, which have yet only been assigned data ranges. When finally the trace is fully instantiated, it is executed against the IUT. In case, the IUT's reactions divert from the originally computed trace, the behavior manager adapts it automatically in cooperation with the data manager and the constraint solver. When a trace has been fully executed, then the test run manager receives the information about the results of the test run and assigns a test verdict.

5.2.4 Component ConstraintSolver

The Component `ConstraintSolver` is formed by the constraint solver ECLiPSe Prolog. In Chapter 3, we discussed the connection of constraint solving and system specifications. How we can make use of this connection for testing purposes became clear in Section 4.4, where we introduced the concept of test oracles for test data selection. In this section, finally, we will discuss the realization of the connection of the constraint solver ECLiPSe Prolog to the test tool BAiT. First, we address a few decisions we made, regarding the test oracle, i.e. its underlying CLP, directly w.r.t. Chapter 3. Then, we discuss the extension of queries to the test oracle for our purposes w.r.t. the design of queries given in Section 4.4. Finally, we will present some technicalities regarding the implementation of the connection to ECLiPSe Prolog.

A Simple Meta Language

In Sections 3.2 and 3.3, we provided a theory for the transformation of μCRL specifications to Prolog by means of the simulation of innermost term rewriting. The principle is to extract parameters of nested terms, simulate rewriting on them and

assign the result to a variable, which is used in place of the nested parameter. In order to evaluate a guard like

$$and(eq(x,5), gt(y,7))$$

we have to transform it to the following query according to our theory:

$$\square \leftarrow \underbrace{eq(\mathbb{N}(x),\mathbb{N}(5),\mathbb{B}(z_1)) \wedge_p gt(\mathbb{N}(y),\mathbb{N}(7),\mathbb{B}(z_2)) \wedge_p and(\mathbb{B}(z_1),\mathbb{B}(z_2),\mathbb{B}(z_3))}_{\text{transformed guard}} \underbrace{\wedge_p z_3}_{\text{evaluation}}$$

In order to make the evaluation work, we have to define two rules in our CLP for the signature of μCRL's boolean algebra, Bool, with $C_{Bool_\mu} = \{T_\mu, F_\mu\}$ with $T_\mu := T$ and $F_\mu := F$:

$$\begin{aligned} t &\leftarrow \top \\ f &\leftarrow fail \end{aligned}$$

The occurrence of the constant t makes a query hold and thus evaluates it to $\top$, while f lets it fail and thus evaluates it to $\bot$. With these two rules given, ECLiPSe Prolog can evaluate the examplary shown query above with the variable z_3 being used as a predicate. However, for the practical realization of test oracles, we decided to do things a bit differently. The Prolog rules for boolean functions should no longer have an explicit result parameter of type $\mathbb{B}$, but should instead be directly interpreted by Prolog. This has the big advantage, that the above-shown example can much easier be encoded in Prolog as:

$$and(\mathbb{B}(eq(\mathbb{N}(x),\mathbb{N}(5))),\mathbb{B}(gt(\mathbb{N}(y),\mathbb{N}(7)))))$$

The approach has, w.r.t. our previously presented theory, also two disadvantages: First of all, boolean functions, which are not used as guards but whose return value is user data, are affected as well, so that their function results can only be evaluated but not be used elsewhere as input anymore. A second issue affects the simulation of term rewriting by the CLP: For boolean functions, not innermost, but outermost, rewriting is simulated. The reason to implement the interpretation of boolean functions like this must be seen in the context of the history of BAiT: While in the beginning, this variant seemed to be the most obvious one, the further work on the theory behind it (cf. Section 3.2) showed that there is room for improvement. In the current version of BAiT, however, we kept the approach as it is, while in future versions, it will be consolidated with the interpretation of non-boolean functions, which have an explicit output parameter.

In order to make the approach discussed above possible, we extended the meta language by some more rules. In particular, we add a rule to strip off typing information

from boolean expressions, redefine and and or locally using the standard Prolog operator and add a number of rules for negated expressions:

$$\begin{aligned}
\mathtt{bool}(x) &\leftarrow x && (5.1)\\
\mathtt{and}(x,y) &\leftarrow x,y && (5.2)\\
\mathtt{or}(x,y) &\leftarrow x;y && (5.3)\\
\mathtt{not_}(\mathtt{bool}(\mathtt{and}(x,y))) &\leftarrow \mathtt{or}(\mathtt{bool}(\mathtt{not_}(x)),\mathtt{bool}(\mathtt{not_}(y))),!. && (5.4)\\
\mathtt{not_}(\mathtt{bool}(\mathtt{or}(x,y))) &\leftarrow \mathtt{and}(\mathtt{bool}(\mathtt{not_}(x)),\mathtt{bool}(\mathtt{not_}(y))),!. && (5.5)\\
\mathtt{not_}(\mathtt{bool}(\mathtt{not_}(x))) &\leftarrow x,!. && (5.6)\\
\mathtt{not_}(\mathtt{bool}(\mathtt{eq}(x,y))) &\leftarrow \mathtt{neq}(x,y),!. && (5.7)\\
\mathtt{not_}(\mathtt{bool}(\mathtt{gt}(x,y))) &\leftarrow \mathtt{ngt}(x,y),!. && (5.8)\\
\mathtt{not_}(\mathtt{bool}(\mathtt{ge}(x,y))) &\leftarrow \mathtt{nge}(x,y),!. && (5.9)\\
\mathtt{not_}(\mathtt{bool}(\mathtt{le}(x,y))) &\leftarrow \mathtt{nle}(x,y),!. && (5.10)\\
\mathtt{not_}(\mathtt{bool}(\mathtt{lt}(x,y))) &\leftarrow \mathtt{nlt}(x,y),!. && (5.11)\\
\mathtt{not_}(\mathtt{bool}(x)) &\leftarrow \mathtt{not}(x),!. && (5.12)
\end{aligned}$$

The negation operator $\neg$ is implemented as a set of rules not_ for a number of cases. Rules (5.4) and (5.5) realize *De Morgan's laws* (De Morgan, 1860). Rule (5.6) handles double negation and rule (5.12) hands the standard negation over to Prolog's negation operator not. Rules (5.8) to (5.11), finally, realize the negation of comparisons, like $x \not\geqslant y$, in an intuitive way, in our example $x < y$. Therefore, these rules make use of some rules, which have already been discussed in Section 2.4.1.

A Preamble for the CLP

The tool BAiT can handle data of different kinds of domains, like numerical or enumerative data. While the domains for numerical data are inherently defined in Prolog, the ones for enumerative data are not. For each enumerative domain, a domain specification must be added to the CLP. If a variable of an enumerative domain is unbound, its domain must be declared. For the prior, we have to define domains as follows:

$$\square \leftarrow \mathtt{local\ domain(<name>(<values>))}$$

For example, the definition of the (also enumerative) datatype $\mathbb{B}$ looks as follows:

$$\square \leftarrow \mathtt{local\ domain(bool(t,f))}$$

The domain of an enumerative variable $x : \mathbb{E}$ is in Prolog defined as x&::E. In order to be able to define the domains of variables as $\mathtt{setdomain}(x,\mathbb{E})$, we define an additional rule

$$\mathtt{setdomain}(v,d) \leftarrow v\&::d.$$

Another issue to prepare in the preamble is the aquisition of value ranges for unbound numerical and enumerative variables in a query. Structured data will not

yet be regarded. ECLiPSe Prolog provides two built-in rules to determine this information for a given variable: print_solver_var and get_integer_bounds. Both rules have a different signature. While print_solver_var returns a list of upper and lower bounds of a possibly nonmonotonous data interval for a given variable, get_integer_bounds only returns two simple numbers, namely the variable's lower and the upper bound. A second complication w.r.t. these rules is the fact, that each of them can fail. In this case, the respectively other rule must be used. In order to simplify things for the queries to the test oracle, we introduce in total three rules, which façade the above-mentioned two ones:

$$\begin{aligned} \mathrm{getboundaries}(\_n, x, xb) &\leftarrow \mathrm{print\_solver\_var}(x, xb) \wedge_p ! \\ \mathrm{getboundaries}(\_n, x, xb) &\leftarrow \mathrm{number}(x) \wedge_p \mathrm{get\_integer\_bounds}(x, x1, x2) \\ &\quad \wedge_p xb :: [x1..x2] \wedge_p ! \\ \mathrm{getboundaries}(\_n, \_x, \_xb) &\leftarrow \top \end{aligned}$$

These rules take three parameters. The first one is an arbitrary number n, which is to be ignored by the constraint solver (thus the underscore) and is later needed to identify the variable, whose boundaries have been determined. The second parameter x takes either a constant value or a constrained variable and tries to determine its boundaries, which are returned with the third parameter xb. The boundaries are in all cases returned as a list. While in the first case, the result of print_solver_var can be used directly, we have to actively transform the result of get_integer_bounds into such a list, which happens using the ::-operator of Prolog in the second rule. If both rules fail, we will leave the boundaries undefined rather than letting a whole query fail for unretrievable variable boundaries. For this, we define the third rule, leaving the boundaries undefined.

Queries to the CLP

As we have already discussed in Section 4.4, queries to the test oracle are based on selected traces from the test cases generated by TGV. During test execution, a trace from the CTG is selected and transformed into a query to the constraint solver. The principle of transforming a trace to a query has already been discussed in Section 4.4 and is here extended by the treatment of enumerative datatypes and the acquisition of variable boundaries.

Let us first introduce a helper rule to represent the initialization of a system with specification $\mathfrak{S} = (L, Var, A, E, (\ell_{init}, \eta_{init}))$ in the CLP. This state is represented as a rule

$$\mathrm{init}(\mathrm{state}(\ell_{init}, \eta_{init})) \leftarrow \top.$$

Now let us consider the structure of queries to the test oracle. Let π be the trace under consideration. Let $\mathbb{E}_i$ be arbitrary domains of enumerations and let $\sigma_0 = \sigma_{init}$. Then this query has the form, which is shown in Figure 5.3. In case, the query holds, i.e. it finally evaluates to $\top$, the selected trace from the CTG is valid w.r.t. the system

$$\underbrace{\bigwedge_{i\in\mathbb{N},v\in\mathbb{E}_i} setdomain(v,\mathbb{E}_i)}_{(1)} \wedge \underbrace{init(\sigma_0)}_{(2)} \wedge \underbrace{\bigwedge_{j\in[1..|\pi|],s(x):=\pi[j]} s(\sigma_{j-1},\sigma_j,param(x))}_{(3)} \wedge$$

$$\underbrace{\bigwedge_{w\in\mathbb{Z},k\in\mathbb{N}} getboundaries(k,w,boundaries(w))}_{(4)}$$

Figure 5.3: Structure of queries to the constraint solver

specification (it is *possible in the system*). In this case, it can be executed after all data elements have been instantiated. This conjunction can be divided into four subformulae, which we will discuss now:

1. In the first subformula, all variables, which appear in the trace and which have an enumerative data domain, are defined according to this domain. This happens using the rule setdomain, which has been defined earlier in this section. The definition of domains as such is tautological, i.e. it never becomes false by itself and thus does not negatively affect the query.
2. Then, the trace under consideration is initialized. This happens by introducing a variable σ_0 and matching it to the initial data structure $state(\ell_{init}, \eta_{init})$, which is provided by init as defined earlier. Initializing a yet uninstantiated variable with the system's initial state also always succeeds, assuming that the types of variables and any provided values properly match.
3. The third part forms a conjunction of summand rule invocations and resembles the actual trace under consideration. This part of the query has already been discussed in Section 4.4. Each step $\sigma_{j-1} \xrightarrow{\lambda_j} \sigma_j$ with $\lambda_j := s(x)$ in this trace is represented by one rule invocation $s(\sigma_{j-1}, \sigma_j, param(x))$. One such rule invocation holds, if
   a) there exists a rule, which syntactically matches the invocation, and
   b) the guard g, which is defined in this rule, holds (cf. Section 3.3).
4. The fourth part finally retrieves information about the intervals, to which numerical variables are limited in this trace. This is done by invoking the rule getboundaries with an index (determining the position of the variable in the parameter list of the according action), the variable itself and an uninstantiated second variable to store the interval. Retrieving a variable's boundaries also always holds (by definition earlier in this section).

The order of subformulae is formed by certain restrictions. The order of the trace initialization (2) and the trace itself (3) is obvious. The domains of enumerative variables (1) must be defined before the first constraint tries to match a particular variable

of this kind. For this reason, these domains are defined first. The intervals of numerical variables (4), however, must be retrieved last. These intervals are restricted by *all* constraints in the trace under consideration, so that the trace must be fully evaluated before the variable intervals are queried.

Interaction between BAiT and ECLiPSe Prolog

ECLiPSe Prolog provides a Java API for the integration of a constraint solver into an application as BAiT. This API has been in detail described in Novello et al. (2005), so that we will only give a short overview on the steps, BAiT has to take in order to benefit from the constraint solver. A constraint solver as ECLiPSe Prolog is a resource-intensive component. Therefore, it has to be setup and shutdown carefully by the embedding application. Both the setup and the shutdown of ECLiPSe Prolog is handled by the test run manager of BAiT. During test execution itself, however, communication with ECLiPSe Prolog is fully handled by the trace manager.

The trace manager and ECLiPSe Prolog interact synchronously via the function `rpc()` of class `com.parctechnologies.eclipse.EclipseConnection`. It takes a query to the constraint solver either as a string or a structure of terms (classes `CompoundTerm` and `Atom`, depending on the structure of the particular term) and synchronously returns a structure of terms. There are also asynchronous ways to interact with ECLiPSe Prolog, but we need all test data before the execution of the test trace, so that asynchronous constraint solving is not an option for our case.

In order to let the constraint solver solve the query for a trace, this query is dynamically generated by the trace manager as shown in the previous subsection. It is generated as a string rather than a structure of terms, since in the latter case, the multiple occurrence of a particular variable within the query cannot be expressed (Novello et al., 2005). The query is then handed over to ECLiPSe Prolog by calling `rpc()` and the result is fetched. This result is then interpreted by the trace manager and subsequently by the data manager as described in the following sections.

The function `rpc()` always only returns the first result from a set of results (ibid), which is in most cases sufficient, in which data of a numerical or enumerative type must be instantiated. However, in cases, where structured data must be instantiated, this behavior gets problematic. While for numerical or enumerative data, ranges are returned by the constraint solver, the solver only returns single solutions for structured types. In this case, BAiT immediately takes this first solution to work on further, rather than collecting more solutions from ECLiPSe Prolog. Even though it is possible to realize such a behavior, it is a tricky undertaking and has not yet been implemented in this version of BAiT.

5.2.5 Component DataManager

The data manager `DataManager` is responsible for encoding and decoding data elements for both the constraint solver and the IUT. Furthermore, the implementation of test data selection algorithms is located in this component. The component

`DataManager` is realized by two components, `Variables` and `Instantiator`, as can be seen in Figure 5.4. Those two components are themselves realized by classes and interfaces, which are organized in one package per component. In this section, we will describe these classes and their interfaces.

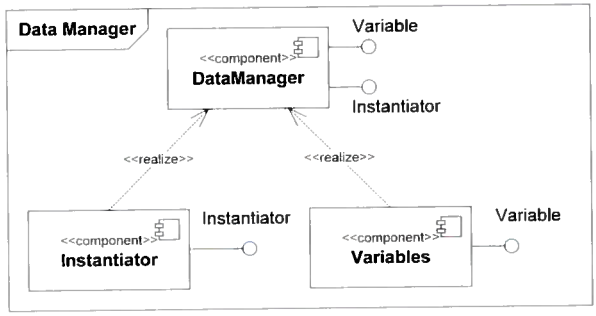

Figure 5.4: Component `DataManager`

Component Variables

The component `Variables` is realized by the classes of `nl.cwi.sen2.bait.variables`, as it is depicted in Figure 5.5. The classes from this package encode and decode values between the constraint solver, BAiT's internal representation and the IUT's data representation. Furthermore, they manage data boundaries for test data selection.

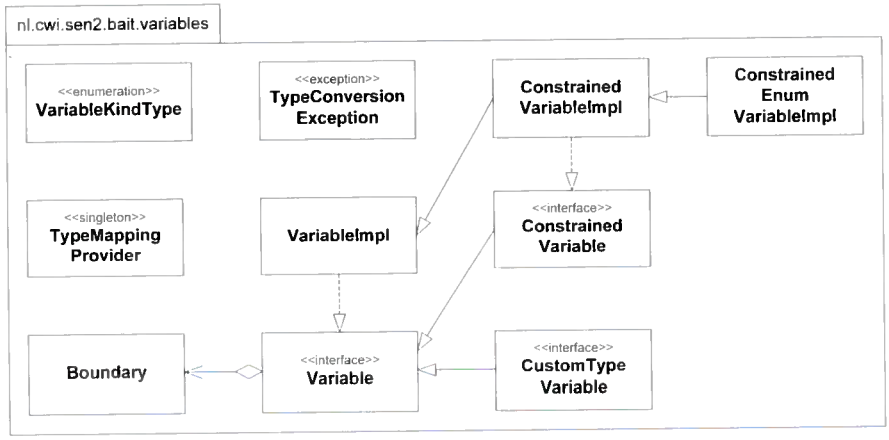

Figure 5.5: Package `nl.cwi.sen2.bait.variables`

Amongst other things, encoding and decoding data between different systems requires a management of datatype mappings. Although the constraint solver is untyped, we have to map datatypes between the specification language (in our case

μCRL) and the IUT's target system (Java). A third datatype is formed by the representation of data elements within BAiT, which happens by the classes of package `nl.cwi.sen2.bait.variables` and classes derived from those.

Let us consider the value $1 \in \mathbb{N}$ as an example: In the representation for the constraint solver, this value is encoded as $\mathbb{N}(1)$, i.e. a numerical 1 attributed with its datatype from the system specification (cf. Section 3.2). For the target system of the IUT, the same value is encoded as an instance of class `java.lang.Integer` with a value of 1. Internally to BAiT, finally, the same data element is encoded as an instance of class `nl.cwi.sen2.bait.variables.ConstrainedVariableImpl` with a value of 1, the (specification) datatype $\mathbb{N}$, an empty variable name and some more attributes.

This management of datatype mappings is provided by the singleton class `TypeMappingProvider`. It allows to add a datatype mapping (method `addTypeMapping()`), which is then globally used in BAiT, and to retrieve information about datatype mappings (methods `get*Type()`) providing the specification datatype or the Java datatype, resp.

In BAiT, we distinguish four kinds of variables. Those kinds of variables are defined in the enumeration `VariableKindType`:

numerical variables, which are handled by class `ConstrainedVariableImpl`,

enumerative variables, handled by `ConstrainedEnumVariableImpl`,

lists and

structured variables. For those two kinds of variables, the class `CustomTypeVariable` serves as an implementation base for codec functionality.

All variable classes implement at least the interface `Variable`. It provides access to a variable's name, datatype, value in the target system and boundaries (implemented in class `Boundary`). Furthermore, it provides functionality to check, whether the variable is actually instantiated and whether a particular data value lies within the variable's boundaries. The interface has been implemented in class `VariableImpl`, which serves as a base class for all general and custom variable classes. Custom variable classes not only inherit from `VariableImpl`, but also have to implement the interface `CustomTypeVariable`. Those custom variable classes realize the representation of structured datatypes or lists, for which not only a particular value in the sense of the target system (i.e. the instance of a Java class, for instance) must be provided, but also access to the variable's fields in their BAiT-internal representation. The appropriate method `getFields()` consequently returns a hashmap of field names to data items of type `Variable`.

Up to now, no data encoding and decoding has been taken place in order to communicate with the constraint solver. The accordant functionality is defined by methods of the interface `ConstrainedVariable` and basically implemented in the classes `ConstrainedVariableImpl` and `ConstrainedEnumVariableImpl`. The difference of the latter lies in its ability to handle enumerative datatypes. `ConstrainedVariable` provides methods for encoding and decoding data for the constraint solver. The encoding method `getPrologEncoding()` returns a Prolog term as a string. This term can either be a variable name for uninstantiated variables or a value for instantiated

ones. For custom variable classes, an implementation of `getPrologEncoding()` would consequently return a term functor and the (partially or fully instantiated) fields of the data structure as arguments of the functor. The method `setTerm()` gets a Prolog term as a parameter and interprets it in order to determine the values or boundaries, which have been computed by the constraint solver for the particular variable.

Comparison to the TCI-CD and TRI of TTCN-3 The TTCN-3 Control Interface, Coding/Decoding Interface (TCI-CD; ETSI, 2003b) provides functionality to handle the encoding and decoding of data between the test execution runtime of TTCN-3 and its environment or the IUT, resp. The codices themselves are implemented in the TTCN-3 Runtime Interface (TRI). We will in the following give a short comparison of the design of the package `nl.cwi.sen2.bait.variables` to the accordant TTCN-3 interfaces.

TCI-CD is responsible for the internal handling of data with the TTCN-3 test execution runtime. It has two base classes: `Type` handling datatype information and `Value` handling data values. This distinction is necessary due to the fact, that in a TTCN-3 test case not only variables, but also datatypes can be defined. Hence, the description of datatypes must also be instantiable within the TTCN-3 runtime.

Test cases for BAiT do not contain any information about datatypes, but only about the occurrence of variables of a particular type. For this reason, we do not (have to) follow a strict distinction of type and value information, and so we chose for the simpler interface `Variable` instead. However, we have instead to handle datatype *mappings* rather than definitions. This happens in the separate class `TypeMappingProvider`.

TTCN-3 `Type`	BAiT
`getDefiningModule()`	n/a
`getName()`	`Variable.getType()`
`getTypeClass()`	`Variable.getKind()` (coarser)
`newInstance()`	`TypeMappingProvider.getBoxedTypeInstance()`
`getTypeEncoding()`	n/a
`getTypeEncodingVariant()`	n/a
`getTypeExtension()`	n/a
`getTypeForName()` (TCI-CD)	`TypeMappingProvider.getUnboxedType()`, `TypeMappingProvider.getBoxedType()`

Table 5.1: Mapping between TTCN-3 type handling and BAiT

The type information methods from TTCN-3 can be mapped to `Variable` and `TypeMappingProvider` as can be seen in Table 5.1. All methods from `Type`, which are not applicable in BAiT, refer to datatype definitions in a TTCN-3 module.

The value management methods from TTCN-3 (TCI-CD) can be mapped to `Variable` as can be seen in Table 5.2. All methods from `Value`, which are not applicable in BAiT, refer to datatype definitions in a TTCN-3 module.

TTCN-3 Type	BAiT
`getType()`	`Variable.getType()`
`notPresent()`	n/a (no omission of values)
`getValueEncoding()`	n/a
`getValueEncodingVariant()`	n/a
`get[Type]()`	`Variable.getValue()`
`set[Type]()`	`Variable.setValue()`

Table 5.2: Mapping between TTCN-3 value handling and BAiT

If one compares those methods of `Variable`, which have a counterpart in TCI-CD, with the complete interface of `Variable`, one finds out that there are still some methods left. Those methods, which have not yet been named here, are necessary for constraint solving and data selection. In addition to the methods named here, `Variable` also supports *variable names* of uninstantiated variables for the constraint solver as well as the management of *boundaries*, which is essential for test data selection. These two aspects are not supported by the TCI-CD representation of datatypes and values.

The TRI also defines classes to support data exchange between the TTCN-3 runtime and an IUT. Such an explicit support is necessary for TTCN-3, since it is designed to be platform-independent. Such a platform-independence was not a particular design goal in the development of BAiT. However, since BAiT is extensible for custom datatypes and communication paradigms, such platform-independence can be achieved up to a certain point.

TRI defines data mappings between two systems: the TTCN-3 runtime and the platform of the IUT. Here, we already have the first difference to BAiT, where *three* systems must be taken into account: Data must be represented internally in the BAiT runtime, and externally for the constraint solver and for the IUT. The second fundamental difference between type mapping by TRI and BAiT is the fact, that TRI defines methods to control data from within a test case. Data in BAiT, however, is fully controlled by the constraint solver and not by the test case logic. For these reasons, datatype mapping as it has been realized in TRI is not applicable for BAiT.

Component Instantiator

A constraint solver cannot always directly assign a particular value to a variable or a parameter of an action in a test case, but compute a possible range of values only. In this case, an algorithm which selects a data value from this range, is needed. Such algorithms are implemented within the package `nl.cwi.sen2.bait.instantiators`, whose classes realize the component `Instantiator` (Figure 5.6).

An instantiator must implement the interface `Instantiator`, which only contains the method `getValue()`. As a parameter, this method gets an instance of a variable (interface `Variable` from the package discussed above) and it selects and returns a value to instantiate the variable.

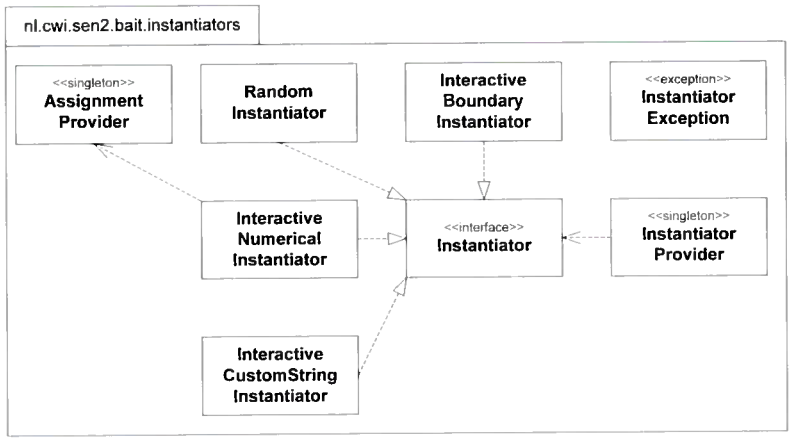

Figure 5.6: Package `nl.cwi.sen2.bait.instantiators`

The underlying data selection algorithm is implemented in the method `getValue()`. In the reference implementation of BAiT, we find four different algorithms implemented in the following classes:

`InteractiveNumericalInstantiator` selects data interactively: The test engineer is provided a prompt and is asked to manually select a value from within the boundaries of the variable.

`InteractiveCustomStringInstantiator` also selects data interactively: The test engineer is provided a prompt and is asked to manually select a value for the variable under instantiation by entering a string. This string has the form of a term as it is used in the specification of the system for data values of the particular type of variable.

`InteractiveBoundaryInstantiator` also selects data interactively: The test engineer is provided a prompt and is asked to manually select the lower or the upper bound of the variable's boundaries.

`RandomInstantiator` automatically chooses a random value to instantiate the variable.

Data selection algorithms are instantiated dynamically using the singleton class `InstantiatorProvider`. This dynamic instantiation of the algorithms allows to configure data selection at runtime.

Up to this point, data for variables is selected on a per-variable-basis and variables are only set into relation with each other within the constraint solver. For some data selection algorithms, it is, however, not sufficient, to only be aware of the value of a single variable. In order to also set variables into relation on the level of data selection algorithms, the singleton class `AssignmentProvider` acts as a scratch pad for all data which is needed to build up this relationship by holding a hashmap mapping a variable name to an object, which can hold an arbitrary item of information.

There is one limitation according to the instantiation of variables: Since we evaluate a trace in beforehand using a constraint solver, the analysis of a variable's limits by *exceeding* them is system-inherently not possible. In that case, the considered trace could not successfully be solved by the constraint solver and would not be executed.

5.2.6 Component BehaviorManager

The component `BehaviorManager` is responsible for selecting single traces from the CTG and executing them in parallel to the IUT. The latter task mainly comprises encoding and decoding of messages for the IUT. The actual implementation of BAiT only supports a procedure-based communication, but can be extended to other communication paradigms, for instance socket-based communication.

The component is realized by two sub-components, `Steps` and `TraceProviders`, as can be seen in Figure 5.7. As in the previously described component `DataManager`, those two components are themselves realized by classes and interfaces, which are organized in one package per component.

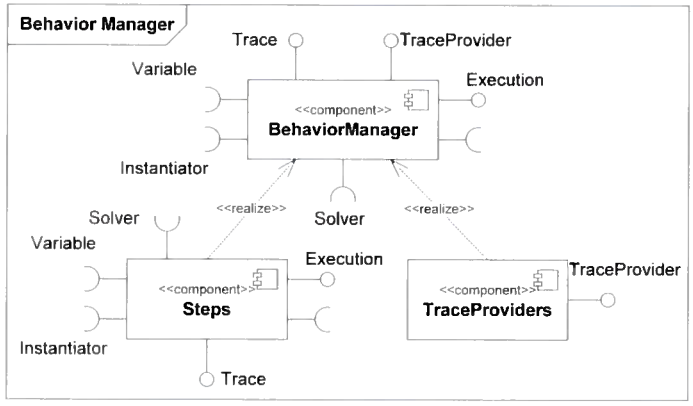

Figure 5.7: Component `BehaviorManager`

Component Steps

The component `Steps` is realized by the classes of package `nl.cwi.sen2.bait.steps` (Figure 5.8). The classes from this package encode and decode complete messages between the BAiT runtime and the IUT. Furthermore, they encode traces as queries to the constraint solver in order to find a solution for the set of parameters in the particular active trace.

Steps Classes representing single steps in a test trace have to implement the interface `Step`. It contains several methods to generally manage an action step in test

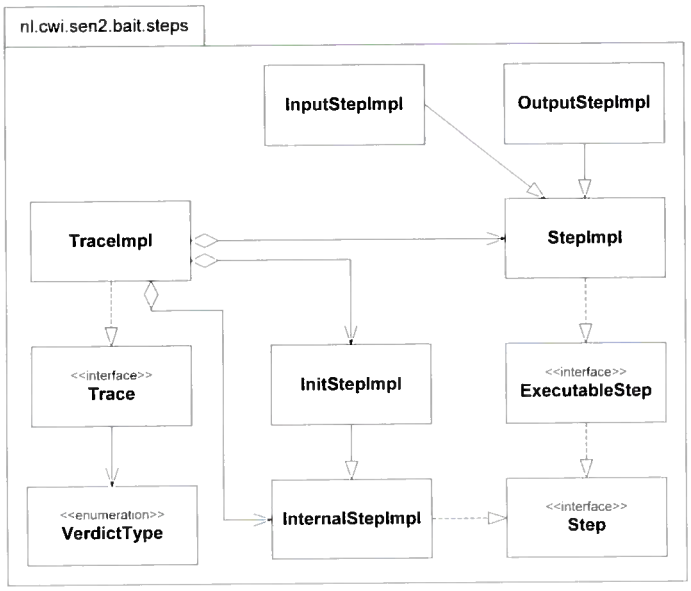

Figure 5.8: Package `nl.cwi.sen2.bait.steps`

execution. These methods get or set the name and parameters of the represented action as well as its place in the trace, which it belongs to. Furthermore, it contains a method `syntacticallyEqual()`, which checks, whether a particular step matches the signature (i.e. action name and parameter types) of another step.

Beyond these methods, a general step implementing `Step` only supports encoding and decoding of data for the constraint solver and a method to instantiate the step's parameters using an instantiator (see Section 5.2.5). In order to build section (1) of a query to the constraint solver (cf. Figure 5.3), method `getVariableDeclarations()` returns a list of enumerative variables and their datatypes. In order to build section (3) of the query, `getStepRepresentation()` transforms the action step into the form given in Figure 5.3. Finally, `getVariableRepresentation()` returns a list of rules of all numerical or enumerative variables, which is necessary to build section (4) of the query to the constraint solver. A result from the constraint solver is parsed and processed in method `setConstraintSolverResults()`.

Steps, which only implement the interface `Step`, are internal steps (class `InternalStepImpl`). These are steps, which do not influence test execution in parallel to the IUT, but are only needed for bookkeeping while solving a trace with the constraint solver. Such actions are τ-steps, but also the *initial step* (class `InitStepImpl`). This initial step does not represent any action, but sets the state vector of the tested system to its initial state. The initial state of the IUT as defined in its specification had been transformed to a rule `init/1`. The initial step, as the first step in any test trace, only generates the according section (2) of the query to the constraint solver (Figure 5.3).

Steps, which communicate with the IUT, implement the interface `ExecutableStep`. It inherits from `Step` and completes it by a method `execute()`. In this method, all the encoding of messages to, and – if applicable for the particular communication paradigm – decoding of messages from the IUT takes place. In the reference implementation of BAiT, we find the class `StepImpl`, which implements `ExecutableStep` and provides the general functionality including encoding and decoding the information about a step for the constraint solver. From this class, `OutputStepImpl` and `InputStepImpl` are derived. While the first contains the code to access the IUT via the Java reflection API, the second class only contains an empty implementation of `execute()`, since processing the reactions of the IUT in the procedure-based setting happens in a different way, as we will see in the next paragraph.

Traces A trace is in principle an ordered set of steps together with a preset test verdict. Traces, like the reference implementation `TraceImpl`, implement the interface `Trace`. It allows to add steps (methods `addInitStep()` and `addActionStep()`) and to execute them step-wise (method `executeNextStep()`). Reactions from the IUT are buffered internally. When a test run diverts from the precomputed test trace, the actual trace is pruned at the last executable step, which matched the test run. Afterwards, the internal input buffer is appended to the set of steps of the trace and finally, the trace is merged with a newly calculated trace. All three operations, pruning, appending pending input and merging of traces, are performed by `merge()`.

The interface `Trace` also contains the method `solve()`, which implements the actual communication with ECLiPSe Prolog. The method builds a query as given in Figure 5.3, sends it to the constraint solver, retrieves the result and distributes this result over the several steps in the trace. Then the steps' parameters can consecutively be instantiated. When the whole trace, i.e. all its steps, is instantiated, it can be executed.

During the test, a test verdict is assigned to a trace. A test verdict is a value from the enumeration `VerdictType`. Any trace is assigned a *potential* test verdict before being executed. This potential test verdict, either Inconc or Pass, had been previously computed by the test generator TGV and has been assigned in the CTG already. When the particular trace could be executed sucessfully to its end, this potential verdict turns into an *actual* verdict. As long as the trace is still under execution, however, the actual verdict remains None. If the trace could not be executed successfully until its end, then one of two things could have happened: The first possibility is, that a failure in the IUT was discovered by the test run, which then ends with verdict Fail. The second reason can be a problem in the test environment, like a broken connection to the constraint solver. In such a case, the test run ends with verdict Error. Those two verdicts, however, are not assigned to a test trace as potential test verdicts.

For a procedure-based test with BAiT, a trace adapter must be derived from `TraceImpl` for each test project, which implements an interface of all possible input actions from the IUT. This interface corresponds to the socket `Execution` in the component diagram (the equally-named interface represents the output steps). All input actions are implemented as methods, which only have to log a call from the IUT.

Protocolling happens by calling `protocolInput()` of the class `TraceImpl`. The method receives the name of the called action as well as a list of actual parameters handed in with the method call. The method itself then instantiates a new object of type `InputStepImpl` as a representation of an input step, and adds it to an internally-handled input buffer.

Comparison to the TRI of TTCN-3 The TRI of TTCN-3 consists of two interfaces. The interface `triCommunication` provides methods to exchange messages with the IUT. The interface `triPlatform` provides timer management, external functions of the TTCN-3 runtime and the possibility to reset the platform adapter (ETSI, 2003c).

We do not want to regard this latter interface further, since BAiT is neither focused on platform-independence nor does it provide any external functions. Timers are supported, but in a rather simplified manner, so that they are not externally controllable.

The interface `triCommunication` provides an abstract level for the exchange of messages with the IUT on either a message- or a procedure-based level. From that perspective, the interface matches the interface `ExecutableStep` of BAiT. However, since BAiT does not focus on platform-independence or independence of the communication paradigm used to exchange information between the tester and the IUT, it is realized a lot simpler. The most obvious difference between TTCN-3 and BAiT in this point is, that TTCN-3 does not use a distinct instance of a system adapter per message sent to or received from the IUT through the interface `triCommunication`. BAiT uses one object of type `ExecutableStep` *per step* in the test case. The reason for this is, that in TTCN-3, test cases together with their logic are programmed and are compiled to Java byte code prior to execution, while in BAiT, the test logic is dynamically developed by interpreting a CTG at runtime. The second difference is, that test data selection and test case parameterization mainly happens at the beginning of a test run in TTCN-3, while in BAiT test data is mainly computed during test execution. Both issues, the dynamic computation of test traces and of test data, require management overhead, which can more easily be handled with the design as it has been developed for BAiT.

Component TraceProviders

The component `TraceProviders` is realized by the classes of `nl.cwi.sen2.bait.traceproviders` (Figure 5.9). The classes from this package implement search algorithms in order to select test traces from the CTG, which lead to Pass or Inconc verdicts. These traces can afterwards be instantiated and executed in parallel to the IUT.

All trace search algorithms must implement the interface `TraceProvider`. It offers methods to initiate a trace search (method `generateNewTrace()`), to access buffered traces (see later; methods `getPassTraces()` and `getInconclusiveTraces()`), and to determine, whether there are more traces to find in the CTG (method `hasMoreTraces()`).

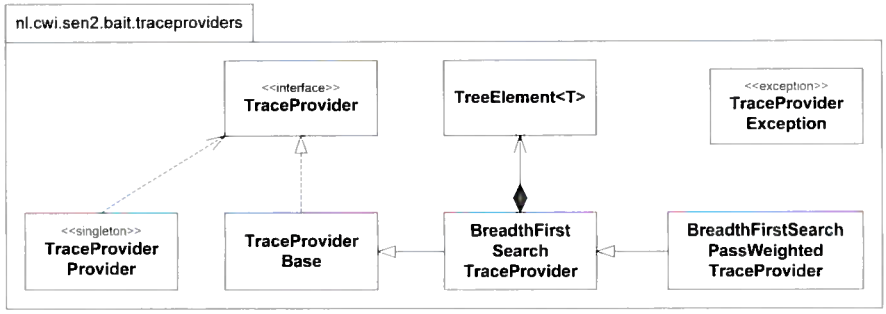

Figure 5.9: Package `nl.cwi.sen2.bait.traceproviders`

Finally, the method `removeShortTraces()` can be used to clean up memory by removing buffered traces, which are shorter than a given length.

A trace search has the goal to find a trace in the CTG which leads to a particular verdict. While searching through the CTG, a trace search algorithm can (and in the reference implementation does) build up a search tree. In this search tree, for instance loops are successively unfolded. The search algorithm stops at the first trace found, which matches the intended goal. In case of the algorithm, implemented in class `BreadthFirstSearchTraceProvider`, this is the first trace, which ends in a Pass or Inconc verdict. In case of the `BreadthFirstSearchPassWeightedTraceProvider`, it is the first trace to a Pass verdict. For this latter algorithm, all traces found, which are leading to Inconc, are buffered for potential later use.

Since unfolding loops and buffering traces for potential use, memory might get an issue for these classes. During a test run, a precalculated trace is executed in parallel to the IUT and the trace provider is only needed in case the IUT diverts from this trace. The newly found trace must then be longer than the yet executed number of steps. The method `removeShortTraces()` helps to free memory by removing all buffered traces in the trace provider, which are too short to still be interesting. In doing so, the aforementioned memory problem is restrained.

A custom provider can be easily implemented by inheriting from the base class `TraceProviderBase` and implementing custom-made methods `hasMoreTraces()` and `generateNewTrace()`. The singleton class `TraceProviderProvider` supports the selection of the trace provider used during a test run.

5.2.7 Component TestRunManager

The component `TestRunManager` is realized by the abstract class `nl.cwi.sen2.bait.RunTest`, which controls the whole test execution and can be considered the only active class in the framework. It implements the four-phased test execution process as it is shown in Figure 5.10.

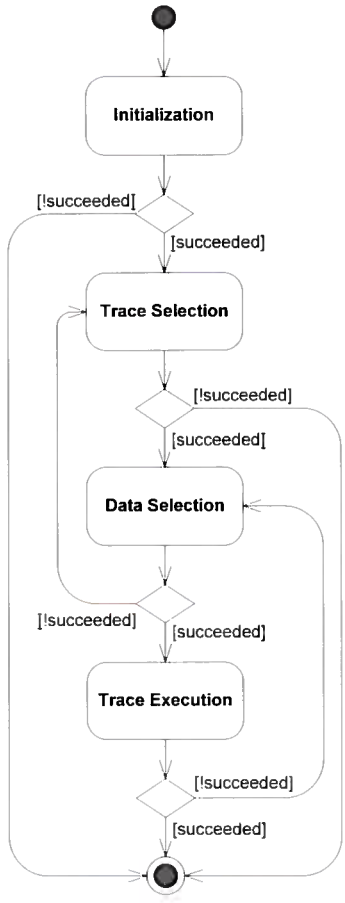

Figure 5.10: The test execution process

The class `RunTest` is abstract. This means, that for every test project, a custom test runner class must be inherited from `RunTest`. This test runner must implement the methods `getIUT()` for test object instantiation and `getAdapter()` for the instantiation of the test adapter. The procedure has been exemplarily shown in Calamé (2007). In the following, we will discuss the different phases of the test execution process.

Initialization In the initialization phase, mainly three things happen. The framework sets some basic properties, described in (Calamé, 2007, Section 4.3), which configure the further test run, and also defines the basic datatype mappings (ibid). While setting the basic properties, an instance of the trace selection algorithm is created.

Afterwards, an instance of the IUT and an instance of the trace adapter are created. We name this instance of the trace adapter further the *trace under execution*. This

trace under execution is attached to the instance of the IUT. In the beginning of test execution, this trace only holds a single initialization step. Having set up the test so far, the first iteration of the trace selection phase is entered. If anything goes wrong during test intialization, test execution terminates with an Error verdict.

Trace Selection Trace selection is performed as has been described earlier in this document. When a trace has been found, the selection algorithm first creates an instance of class `TraceImpl` as the internal representation of traces in BAiT. Then, the test run manager tries to merge this trace to the trace under execution. The traces do not replace each other, since then the link to the IUT as well as any information about already executed steps would be lost. Merging succeeds, if the new trace starts with the same sequence of actions as the trace under execution has executed so far. Also the action parameters must match. The latter requirement already interferes with the data selection phase.

If a trace was successfully selected, data for yet uninstantiated action parameters is selected in the next phase. If a trace did not match the trace under execution, trace selection iteratively goes further until there are no more traces in the test case to select from. In this case, test execution terminates with the verdict Inconc.

Data Selection In the data selection phase, the yet uninstantiated parameters of actions in the trace under execution are instantiated. Therefore, the constraint solver is invoked to determine the intervals of all variables in the trace. Then, the first variable is instantiated by the selected data instantiator and afterwards, the constraint solver is invoked again, the next variable is instantiated and so on.

Invoking the constraint solver for each of the variables again, helps refining the intervals of remaining variables before their instantiation. In many cases, these intervals reduce to a single value so that only a few variables have to be instantiated by the instantiation algorithm. Another approach would be, to invoke the constraint solver twice, once before and once after having instantiated *all* variables. This, however, leads to a less precise variable instantiation and thus to more cases, in which a trace must be adapted before being executed further. If all parameters could be instantiated, the trace execution phase is entered. Otherwise, the trace selection phase is re-entered.

Trace Execution When a trace has been successfully merged with the trace under execution, the actual execution begins. Executing a trace means, that a program pointer iterates through the ordered set of steps in the trace. Depending of the actual step under execution, one of the following happens:

- If the actual step is an *initialization* or an *internal* step, it is ignored and the program pointer is increased.
- If the actual step is an *input* step, it is matched against the oldest element from the input buffer. If the element matches, i.e. the action name and parameter

values are expected, the program pointer is increased and execution goes on. If the input buffer is empty, the system waits for a certain time for input to match before timing out. If the oldest step does not match or execution times out, the trace under execution is adapted.

- If the actual trace is an output step, it is first checked whether the input buffer is empty. If it is empty, the step is executed, otherwise the trace under execution is adapted.

If the trace under execution was executed successfully to its end, test execution terminates with the verdict, which had been assigned to the trace (Inconc or Pass). If the trace under execution must be adapted, it is cut at the position of the program pointer. Then, all input steps from the input buffer are appended to the remaining stub. Invoking the constraint solver verifies, whether the trace as it has yet been executed, is valid regarding the system specification. If it is invalid, test execution terminates with the verdict Fail. If it is a valid trace stub, the trace selection phase is re-entered.

Before in this re-entrance of the trace selection phase new traces are selected from the test case, it is checked whether the trace under execution could be executed to its end using other values for action parameters (data selection). Then, stored traces to Inconc are checked for applicability and finally, trace selection searches for a new trace to Pass.

5.3 Related Frameworks

In this section, we want to position our tool towards other, well-known test execution frameworks.

5.3.1 TTCN-3

TTCN-3 (Grabowski et al., 2003; Willcock et al., 2005) is a system-independent test description language. It allows the separation of test code and test data to reuse once written code. TTCN-3 is used for the automatic execution of conformance tests, not for their generation.

TTCN-3 code is compiled prior to test execution. This is a difference to our approach, that interprets test cases during execution. The IUT is bound to the TTCN-3 runtime by platform and system adapters, which have about the function of the proxy object in our (simpler) framework. In principal it is even possible to connect BAiT to the TRI/TCI interfaces of TTCN-3. In TTCN-3, test data is either instantiated statically prior to execution or constructed dynamically during test execution, while in our approach, data is only constructed dynamically.

Even though binding a constraint-solver to the TTCN-3 runtime is in principal possible, its sense would be rather limited. Since all the trace and data selection would happen in a constraint-solver proxy, TTCN-3 would just get a next step for execution

and pass it on to the system adapter, receive an event from the IUT and pass that one on to the constraint-solver proxy. This is exactly, what our test execution tool is doing *without* TTCN-3 in between.

However, there would be possibilities to generate TTCN-3 code that is able to adapt its own execution. Since TTCN-3 is a rich language, all facilities necessary to do so (like loops and conditional branching) are available. There are two possibilities to generate TTCN-3. The first one is a generation of TTCN-3 from the TGV test cases and the test oracle. In the presence of loops in the test cases, this approach would quite likely not terminate. If those loops are unfolded during test execution, the IUT terminates mostly within finite time, so that also test execution terminates (even though this cannot be guaranteed, either). So unfolding these loops seems to be an option, however, if this happens prior to interaction with the IUT, it will limit the ability of test cases to adapt to unexpected system behavior induced by a nondeterministic specification.

Another idea would thus be to generate TTCN-3 code directly from the system specification, which defines the logical structure of the system. The result would be a *mirrored* system, that would serve as the tester of the real system. Since the *whole* system and not only parts of it would be mirrored, a limitation of test cases by test purposes would in this case not be possible anymore.

5.3.2 xUnit

Unit Testing Frameworks (xUnit) (`www.xprogramming.com/testfram.htm`) exist for a variety of different system platforms. The "x" in xUnit stands for an arbitrary prefix, depending on the particular framework. A collection of these frameworks has comparatively been described in Calamé (2003).

xUnit is based on the ideas of agile software development and a test-first approach. Test-first means, that the specification of a system is given as a set of test cases. For this reason, Pass and Inconc verdicts are not distinguished.

There is, in most cases, no separation of test code and test data; however, some frameworks support this, e.g. JXUnit for Java as an extension to the standard JUnit. System requirements are stated in `assert...()` methods. Such a method fails immediately, if its requirement is not met by the IUT. In this case, the test case fails; an adaptation of the execution is not possible in this way.

This leads to the conclusion, that if we want to profit from the simplicity of xUnit test cases, we cannot provide test case adaptation in contrast to our framework. Since the basic language of a particular xUnit framework is rich enough to provide loops and conditional branching, the same holds as for the TTCN-3 test case generation discussed in the previous subsection. However, using an xUnit framework in this way would not have any advantages towards just using the framework's basic programming language.

5.3.3 STG

Symbolic Test Generation (STG) is a test generation approach described by Clarke et al. (2002) and more detailed by Jéron (2004), which is implemented in the equally named toolset (`www.irisa.fr/prive/ployette/stg-doc/stg-web.html`) and which consists of a test generator and an execution framework. Like TGV, on whose ideas it is based, the input to STG are a system specification and a test purpose. Since STG works on a symbolic rather than an enumerative level, it is not necessary to generate their state spaces. Both the system specification and the test purpose are thus given as IOSTSs. The specification language used for STG is a proprietary one, which is based on the ideas of the specification language IF. A difference to TGV and to our toolset is, that test purposes may contain guards.

When generating the test case, STG – like TGV – produces the synchronous product of both the system specification and the test purpose from which it then selects test cases. These test cases still contain guards and uninstantiated variables. As its output, STG generates a tester, which is executed in parallel to the IUT. The tester instantiates the variables on the selected test trace, based on output of the *Lucky* solver, taking into account nondeterminism of a system specification. To the best of our knowledge, test trace selection in STG takes place already before test execution, such that the adaption to the system's nondeterminism happens on the level of test data, but not on that of traces. BAiT, on the other hand, supports adaptation with respect to both behavior (i.e., trace) and data.

5.3.4 Qtronic

TTCN-3 is supported by several commercial tools, which can execute or generate test cases. One of these tools is Qtronic (`www.conformiq.com/qtronic.php`). This tool supports on-the-fly execution of test cases as well as the generation of TTCN-3 code. The input for Qtronic are UML state charts with their behavior specified in Java (the combination is named QML Conformiq). Additionally, models can also be specified in Lisp. There exist interfaces for test adapters to C++ and Java.

The tool examines a model and either immediately executes a test on the IUT or generates a TTCN-3 test case. On-the-fly testing can either be random or directled by coverage criteria (condition coverage, branch coverage as well as transition or state coverage on the level of UML). Testing with Qtronic (both on-the-fly and test case generation) does not necessarily terminate, but can be guided by use cases. The documentation of Qtronic leaves open, how test cases can be parameterized with custom data other than that needed for the aforementioned coverage criteria or boundary value analysis.

Chapter 6

BAiT in Action

> Um zu erkennen, ob das Bild wahr oder falsch ist, müssen wir es mit der Wirklichkeit vergleichen. [...] Ein a priori wahres Bild gibt es nicht.
>
> *(Ludwig Wittgenstein)*

In the course of the previous chapters, we had developed a theory for testing reactive systems with data using data abstractions, enumerative test case generation and constraint solving. Furthermore, we have extended this theory to the execution of parameterized test cases and have developed a tool, BAiT, which supports our testing approach. In this chapter, we will evaluate the use of this tool at hand of two case studies.

The first case study regarded in this chapter is formed by testing an ATM. It is an academic example, which evaluates the possibility of BAiT to support the selection of interdependent data and to adapt to unforeseen reactions of the IUT during a test run. The ATM is initialized with a Personal Identification Number (PIN) and a certain balance. Then the user selects an amount of money and the ATM hopefully returns this amount in a certain denomination of bank notes. First of all, the amount of money chosen, the balance of the bank account and the money emitted by the ATM depend on each other. This part is thus a show case for data selection in BAiT. Finally, the denomination chosen by the ATM is specified in a way, that we cannot clearly foresee the ATM's exact reaction on a certain user input. This issue adds the behavior adaptation to the first case study.

The second case study shows the use of BAiT in an industrial context, testing a part of the webbrowser *Mozilla Firefox* (Calamé and van de Pol, 2008). In this case study, we test the web page rendering capabilities of a real-life application. In order to do so, we developed a wrapper around the Firefox rendering component *Gecko* and specified a fragment of the CSS box model of the World Wide Web Consortium (W3C). This wrapper façades Gecko and serves as the IUT during the test with BAiT.

In the remainder of this chapter, we will discuss the two case studies in detail. We will describe the ATM in Section 6.1 and the Firefox case study in Section 6.2.

6.1 A Behavior-oriented Case Study: ATM

This section serves as a tutorial for the test generation and execution process of BAiT. As an example, we present a simple ATM. We will first model the system and after-

wards discuss in detail the inputs, outputs and general steps to be taken for test generation and execution.

6.1.1 Automatic Teller Machine

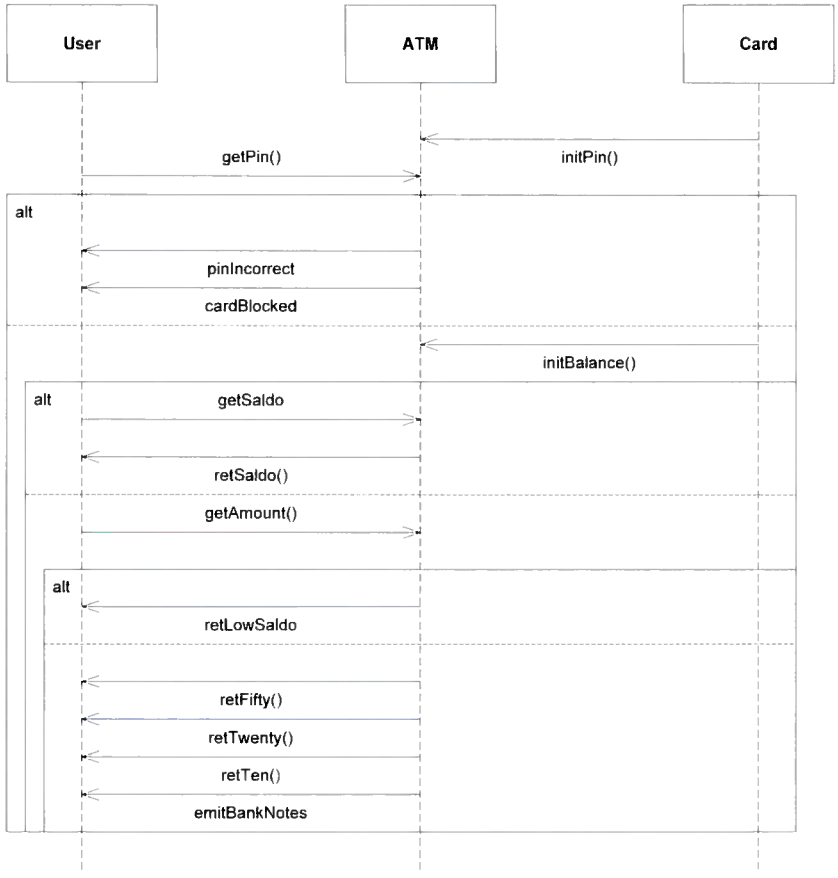

Figure 6.1: Components of the ATM

The system that we use as our case study consists of several components whose communication is shown in Figure 6.1. The main component is the ATM itself. It is a reactive system in an environment, which contains a user of the machine, and the user's bank card. While the bank card communicates unidirectional with the ATM, the user interaction is a bidirectional communication.

The full specification of the ATM is shown in Figure 6.2 as a UML state chart. Inputs to the system are attributed with a question mark, outputs from the system with an exclamation mark. When the user inserts his card, the user's PIN is initialized. The

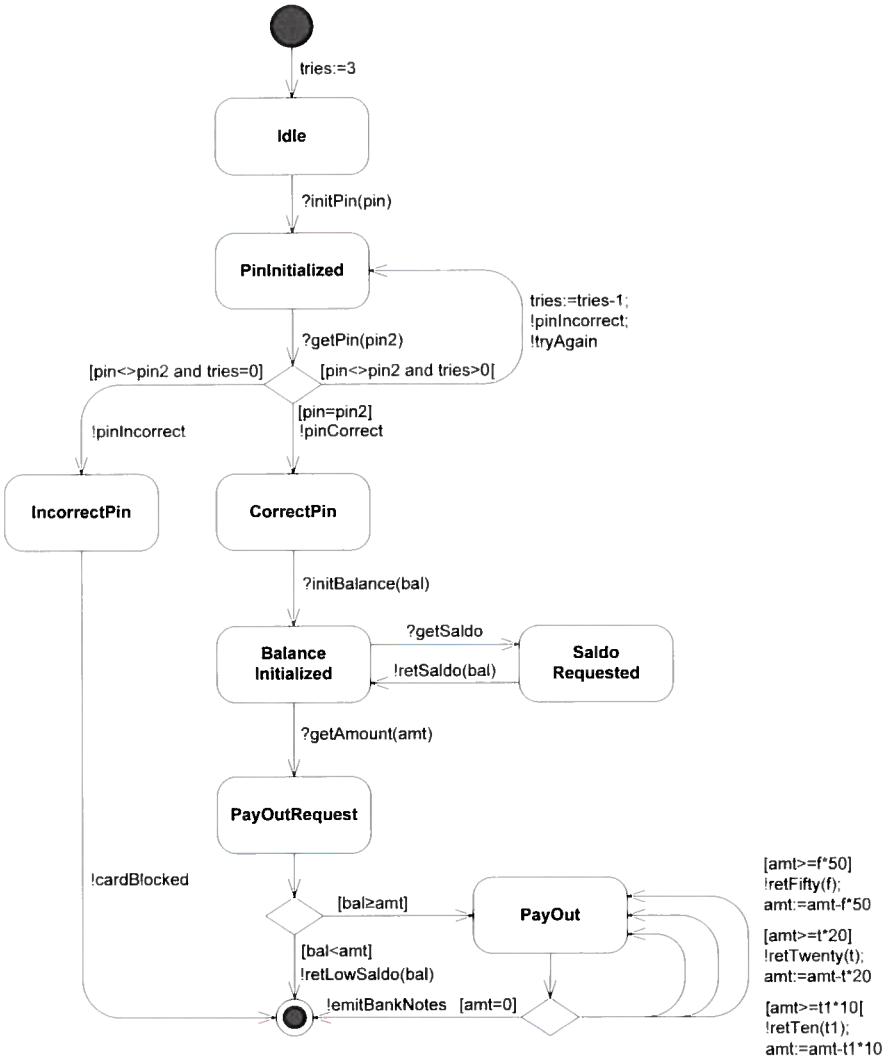

Figure 6.2: Specification of the ATM

user then has to enter the correct PIN. If he succeeds, the ATM sends the message `pinCorrect` and the process can go on; otherwise the message `pinIncorrect` followed by `tryAgain` or – after the third mismatch – `cardBlocked` is sent.

If the user has entered the correct PIN code, his bank account balance is initialized and the user may choose either to review his saldo or to withdraw a particular amount of money. When he chooses the latter option, he either gets the message `retLowSaldo`, if he asked for more money than was actually on the account, or the machine starts paying out.

Paying out money is left nondeterministic in this specification: The machine prepares a certain amount of 50€, 20€ and 10€ bank notes for emission and emits them when they sum up to the requested amount of money. Hereby it is left open in which order and how many of the single bank notes will be emitted. This decision is made later during the implementation of the ATM and will serve as the example for the adaptation of test execution.

6.1.2 Test Generation and Test Execution

In Section 5.2.2, we have already provided a short discussion of the necessary artifacts for test generation and execution with BAiT:

- System Specification
- Test Purpose
- Proxy Classes
- IUT

We will now shortly describe these artifacts. For more detail, we refer the reader to Calamé (2007).

The Test Purpose

In our test, we want to validate that after entering the correct PIN code, we will eventually get some money (even more precisely: we do receive some 10€ bank notes). This scenario is formulated as the test purpose in Figure 6.3. It accepts the occurrence of `emitBankNotes` after entering a correct PIN code (sequence `getPin`, `pinCorrect`) and `retTen`, the preparation of 10€ bank notes for emission. The occurrence of `pinIncorrect` in test traces is refused, i.e. it is not in the focus of this test.

A test purpose does not define the complete trace, as it would be executed during test execution. It rather defines relevant actions in the order, in which they should appear during the test (in our case: `retTen` after `pinCorrect`). Those actions, which are missing in the test purpose, but are relevant during execution, like `getAmount` in our example, are automatically completed during test generation.

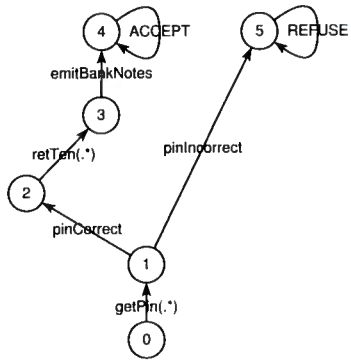

Figure 6.3: Test purpose for the automatic teller machine

Generation of Test Cases and the Test Oracle

After having specified the IUT and having set the test purpose, test generation can start. It is a process, which is mainly – in many cases fully – automated, and which has already been discussed in depth in Chapter 4 with an exemplary case study in Section 4.5.

Simultaneously with the generation of test cases, the test oracle is generated. This generation is based on the theory from Chapter 3 and has been worked out in Section 4.4.

Proxy Classes

The last step before test execution is the implementation or generation of two proxy classes between the generic part of BAiT and the IUT. At this point, we will concentrate on the implementation of the ATM case study. Details on the implementation of the proxy classes are given in Calamé (2007).

Implementation of the IUT The ATM is realized as the (procedure-call-based) Java classes `CATM` and `CATM_faulty` (Figure 6.4, package `atmv3`), which are based on two interfaces. The interface `IATM`, which is implemented by `CATM` and `CATM_faulty`, declares those actions which serve as *input* to the ATM, while `IATMUser` declares the *output actions*. In order to realize the bidirectional communication between the ATM and its environment on a procedural level, the ATM has been realized following the *Observer Pattern* described by Gamma et al. (1995). The number of possible subscribers to emitted events is in our case limited to one. There are two additional methods `attach()` and `detach()` in `IATM` to subscribe a component (i.e. the tester) to or unsubscribe it from the events emitted by the ATM.

The interface `IATMUser` must be implemented by the tester, who must also be attached to the IUT. This leads us to the implementation of the test adapter `ATMProxy` from

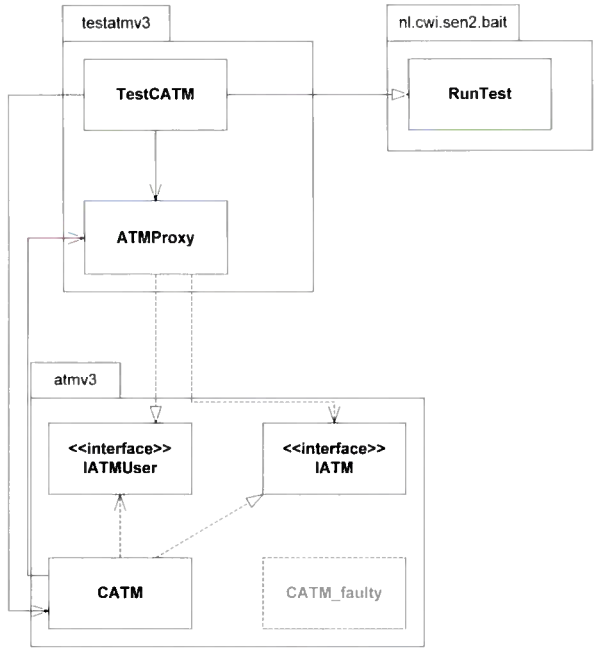

Figure 6.4: IUT (classes `CATM` and `CATM_faulty`) and tester

Figure 6.4. `ATMProxy` implements the interface `IATMUser`, i.e. an implementation is provided for all output actions of the IUT. This implementation does nothing more than logging the action name and all actual parameters received from the IUT.

With `CATM` and `ATMProxy` we have nearly all necessary ingredients to run a test. What we are still missing is the initialization of the IUT and the creation of a proxy object. Therefore, we have to implement or generate the class `TestCATM`, which initializes both the IUT and its proxy. Furthermore, it may define basic settings for the test, like a mapping between datatypes in the specification language and in Java.

6.1.3 Test Execution

After having compiled the IUT and the two Java classes on the tester side, we can now execute the test cases. For this case study, we implemented two mutants of the ATM. `CATM` works correctly, returning 50 €, 20 € and 10 € bank notes, such that the customer not only receives notes of the largest denomination, but also of the smaller ones. This means, that a request for 100 € will result in one bank note at 50 €, two at 20 € each and one at 10 €. The faulty implementation of the ATM, `CATM_faulty`, does not return enough bank notes of the smallest denomination.

When we now execute the test for the correct ATM, we will first be asked to define the values for some variables:

```
Pin in {[-Infinity .. Infinity]} => 5
Bal in {[0 .. Infinity]} => 1000
Amt in {[0 .. 1000]} => 100
```

This happens, since in the default settings test data is instantiated interactively – this could be automated. After entering the values from above, the test will be executed:

```
May 25, 2007 3:09:56 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Trying: [init],initPin(5),getPin(5),pinCorrect,                \
    initBalance(1000),getAmount(100),[tau],retTen(10),emitBankNotes
May 25, 2007 3:09:56 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Expected: retTen(10), received: retFifty(1) -> Trace failed,   \
    trying alternative.
May 25, 2007 3:09:56 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Executed so far: [init],initPin(5),getPin(5),pinCorrect,       \
    initBalance(1000),getAmount(100),[tau],retFifty(1),retTwenty(2), \
    retTen(1),emitBankNotes
May 25, 2007 3:09:56 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Trying: [init],initPin(5),getPin(5),pinCorrect,                \
    initBalance(1000),getAmount(100),[tau],retFifty(1),retTwenty(2), \
    retTen(1),emitBankNotes
May 25, 2007 3:09:56 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Test finished; executed: [init],initPin(5),getPin(5),          \
    pinCorrect,initBalance(1000),getAmount(100),[tau],retFifty(1),   \
    retTwenty(2),retTen(1),emitBankNotes
May 25, 2007 3:09:56 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: The test case ended with verdict PASS.
```

The first thing to remark is, that both the parameters for `initPin` and `getPin` have been instantiated, even though we only provided one value for them. The reason is, that the variable instantiator only tries to instantiate variables, which are not yet defined and whose values cannot be derived from other variables. Since the parameter for `getPin` could be derived from the parameter for `initPin` – they have to be equal for the test to result in a Pass verdict – it is silently instantiated and the test engineer is not asked again.

The second thing to remark is, that the tool first tries to execute the shortest trace to Pass in the test case (first `INFO` line). This fails (second `INFO` line), since the implementation returns one 50€ bank note as explained earlier, instead of ten 10€ notes. Thus test execution must be adapted. In the third `INFO` line, one can see the events sent to and received from the IUT up to now, including the yet unprocessed actions `retTwenty(2)`, `retTen(1)` and `emitBankNotes`. In the next `INFO` line, the alternative trace is shown, which will be executed further. Since it matches exactly with the results received from the IUT, the test ends with a Pass verdict (last line).

Adapting test execution to the output of the IUT does not necessarily lead to a passing test, as we can see below. Adaptation does not lead to Pass here, since the constraint to receive enough money from the ATM is violated in the test of `CATM_faulty` and thus the test ends with a Fail verdict.

```
May 25, 2007 3:06:19 PM nl.cwi.sen2.bait.steps.TraceImpl merge
FINE: Pruning planned test trace.
May 25, 2007 3:06:19 PM nl.cwi.sen2.bait.RunTest findTrace
FINER: Examining: [init],initPin(Pin),getPin(PinUser),pinCorrect,    \
    initBalance(Bal),getAmount(Amt),[tau],retTen(Twe),emitBankNotes
May 25, 2007 3:06:19 PM nl.cwi.sen2.bait.steps.TraceImpl solve
FINEST: init(G1),initPin(G1,G2,lparam(nat(Pin))),[...]
[Pin in {[-Infinity .. Infinity]} => 5]
May 25, 2007 3:06:21 PM nl.cwi.sen2.bait.steps.TraceImpl solve
FINEST: init(G1),initPin(G1,G2,lparam(nat(5))),[...]
[...]
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Trying: [init],initPin(5),getPin(5),pinCorrect,                \
    initBalance(1000),getAmount(100),[tau],retTen(10),emitBankNotes
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.RunTest executeTest
FINE: initPin(5) -> OK
[...]
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.RunTest executeTest
FINE: retTen(10) -> NOT OK
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Expected: retTen(10), received: retFifty(1) -> Trace failed,   \
    trying alternative.
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Trace deviated; executed so far: [init],initPin(5),getPin(5),  \
    pinCorrect,initBalance(1000),getAmount(100),[tau],retFifty(1),   \
    retTwenty(2),retTen(0),emitBankNotes
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.steps.TraceImpl merge
FINE: Pruning planned test trace.
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.steps.TraceImpl merge
FINE: Adding retFifty(1) to trace stub.
[...]
May 25, 2007 3:06:26 PM nl.cwi.sen2.bait.steps.TraceImpl merge
FINE: Adding emitBankNotes to trace stub.
May 25, 2007 3:06:27 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: Test finished; executed: [init],initPin(5),getPin(5),          \
    pinCorrect,initBalance(1000),getAmount(100),[tau],retFifty(1),   \
    retTwenty(2),retTen(0),emitBankNotes
May 25, 2007 3:06:27 PM nl.cwi.sen2.bait.RunTest executeTest
INFO: The test case ended with verdict FAIL.
```

6.2 A Data-oriented Case Study: Mozilla Gecko

In this section, we want to apply conformance testing to testing rendering engines of web browsers. In state-of-the-art web design, content and design are kept separate from each other. Content is defined in the Hypertext Markup Language (HTML), while the design is specified in a Cascading Style Sheet (CSS), which we will describe in Section 6.2.1. When a web page is rendered, the information from the CSS is used to position elements of content on the rendered page. If a web document has a complicated structure, rendering algorithms can turn out to be erroneous, leading to "broken" web pages with mispositioned elements. Rendering a modern web application, whose appearance is dynamically changed on the client side using script languages, like web applications based on *Asynchronous JavaScript and XML* (AJAX; Zakas et al., 2006), is even more demanding for rendering engines. Performing a sufficient conformance test in this context is tedious, so that an automated solution is preferable. In this paper, we present a feasibility study for automated testing of rendering engines using the test tool BAiT.

Here, we validate the applicability of BAiT (Calamé, 2007; Calamé et al., 2007a), a blackbox test execution tool for nondeterministic, data-oriented reactive systems, to test the rendering engine of a web browser w.r.t. the positioning of boxes in the CSS box model. Boxes in HTML are entities like for instance a complete HTML document (`body` element) or a paragraph (elements `p` or `div`), which contain content or other boxes and which are positioned either absolutely or relatively to each other. The box model is part of the W3C CSS Specification (W3C, 2007, Section 8).

We present a feasibility study, applying conformance testing using BAiT to the rendering engine *Gecko*[1], which is used by the open source web browser *Mozilla Firefox*[2]. In order to perform the tests, we formalize the CSS specification and design test purposes. Furthermore, we implement a wrapper component between the tester and *Gecko* in order to achieve a mapping between an action-oriented specification and the document-oriented rendering engine. For several setups of web pages, we then automatically generate parameterizable test cases. Those test cases can be instantiated with varying data settings for the positions of boxes, so that they are reusable for different page layouts. Then, the test cases are executed automatically against the test wrapper and the results retrieved from *Gecko* are interpreted in order to automatically assign verdicts.

Related Work There exists a number of static test suites for the rendering capabilities of web browsers. Each of those test suites consists of a set of HTML and CSS documents with different page layouts. The most well-known one is probably the *ACID 2 Test*[3]. It tests web browsers for their full compliance to the actual version of CSS by rendering a web page with a vast amount of CSS features enabled.

[1] `http://www.mozilla.org/newlayout/`
[2] `http://www.mozilla.com/en-US/firefox`
[3] `http://www.webstandards.org/action/acid2/`

Another set of test suites for the standard compliance of web browsers are the *W3C Cascading Style Sheets Test Suites*[4]. Here, again, we have a set of static documents, which test rendering capabilities for distinct features of CSS. Finally, Mozilla Firefox itself provides a set of static layout regression tests[5], which can be run in debug builds of the software.

Most of the named test suites are, however, not automated. In fact, the files in the test suites, i.e. the test cases, have to be loaded into the browser and then the result of rendering the page has to be visually assessed. This process is not automatic at all, neither on the level of test case generation, nor on that of test execution. This means, that a certain amount of test cases has to be designed and executed manually and the results have to be visually evaluated. This process is time-consuming compared to an automated test process, where test cases – or at least test data – is generated for a number of standard and critical cases. In this case, the number of test cases to be generated and executed can be optimized in order to reduce the absolute number of test cases. The regression tests of Mozilla Firefox are at least automated on the level of test execution, however, they are still founded on a static set of test cases.

The approach, which we propose in this section, provides not only an automated test execution and evaluation of rendering results for a fragment of the CSS features, the box model, but also an automatic generation and variation of tested web page layouts. We chose this fragment, because rendering results, i.e. the position of a box, can be objectively measured (in pixels) rather than having to be visually assessed. The approach of a fully automatic test case generation and execution has the advantage regarding the other named approaches, that the executed test cases can cover more variation w.r.t. data parameters (i.e. the position of boxes), but also regarding the structure of the rendered web pages. The first issue enables us to reuse equally structured test cases (i.e. web pages) and by that to reduce the number of generated test cases. The latter allows us to test rendering web pages with a different interrelation of elements and by that cover a larger variety of possible failures in the IUT.

We are aware of several case study reports of model-based testing, concerning topics like the Conference Protocol (Belinfante et al., 1999), the Storm Surge Barrier in the Netherlands (Geurts et al., 1998), smart card applications (Clarke et al., 2001), the telecommunication sector (Born et al., 2004) or – vertical to our work – the generation of test purposes for the Session Initiation Protocol (Aichernig et al., 2007). To the best of our knowledge, however, this is the first application of model-based automatic test generation and testing techniques to document-centered applications, esp. to HTML rendering engines.

6.2.1 The Test Environment

The test environment for our case study consists of two main components, which we will introduce in this section. On the one hand, we have the tester, which controls

[4] `http://www.w3.org/Style/CSS/Test/`

[5] `http://www.mozilla.org/newlayout/doc/regression_tests.html`

the run of the experiment. On the other hand, we have the IUT. This is the object under consideration, which we will actually be testing throughout the case study. Finally, we will give an introduction to the CSS box model.

Firefox and Cascading Style Sheets

Firefox Mozilla Firefox is a stand-alone web browser, which has its roots in the Netscape Communicator from the 1990s. Most of its code was put under an open-source license in 1998 and founded the basis for the Mozilla Suite, from which Firefox arose as a stand-alone browser in 2004.

A web browser reads and interprets HTML structured data to display web pages. The task of actually displaying is carried out by a rendering component. While loading a web page, this component incrementally builds up a Document Object Model (DOM) tree of the HTML document to be displayed together with declarative layout information. Mozilla Firefox uses the renderer *Mozilla Gecko*, whose version 1.8.1 we consider in this section as the IUT.

Cascading Style Sheets In the 1990s, a declarative stylesheet language was developed for structured documents in order to properly divide the content of a web page from its design. Currently, the de-facto standard for CSS is in version 2.1 (W3C, 2007). This version is currently not fully supported by all web browsers, including Mozilla Firefox. This issue, however, does not affect our case study.

The CSS design definition for a web page can be provided in three different ways: as an external CSS file, which is linked to the HTML file of the web page, inline the HTML web page and inline a particular element of the web page. In the first two cases, a stylesheet is a collection of blocks of the following form:

```
element.class#id {property_1: value_1; ...; property_n: value_n; }
```

The literal `element` denotes one of the possible elements of HTML (W3C, 2002), like – for simple boxes – `div` or `span`. The literal `class` denotes a user-defined specialization of an HTML element, while `id` denotes a user-specific identifier for a particular occurrence of this element in an HTML document. For each of these elements, we can now define pairs of properties and values.

Figure 6.5 shows a small example: Boxes of class *warning* are rendered with a red border and red text. The one warning box, with the identifier *warning1*, additionally has the text in *italic*. Since this box is a warning box, too, it also takes over all properties from the warning boxes (red border and red text). As a result, the shown HTML code fragment is rendered as two red boxes, embedded into each other, of which the inner box has italic text.

If a CSS design definition is provided in a separate file, it is linked to the web page by using the `link` element of HTML in the following way:

Web page content (HTML):

```
...
<div class="warning">
  ...
  <div class="warning"
       id="warning1">...</div>
  ...
</div>
...
```

Web page layout (CSS):

```
div.warning {
  border-style: solid;
  border-color: red;
  color:red;
}
div.warning#warning1 {
  font-style: italic;
}
```

Figure 6.5: Two differently formatted boxes

```
<link rel="stylesheet" type="text/css" href="mycss.css" />
```

CSS information can also be provided inline the HTML document by nesting it in a `style` element. Inline CSS on HTML element level omits the block structure from Figure 6.5. The definition of the outer box from the figure using CSS inline the element itself, as we will study it here, looks as follows:

```
<div style="border-style:solid;border-color:red;color:red;"></div>
```

The CSS Box Model

In our case study, we will regard the positioning of `div`-boxes by the Gecko rendering engine. Therefore, we have to regard both the dimensions of a box as well as other parameters, which determine the box's position or its distance to other elements on a web page. The interrelation of boxes on a web page is defined by the CSS box model (W3C, 2007, Section 8), which we will briefly introduce here.

The dimensions of a `div`-box are essentially determined by two CSS properties: `width` and `height`. Furthermore, a minimum width and height can be defined as well as their maximum counterparts.

In addition to its width and height, a box also has a number of distances to contained or surrounding content. Those settings are displayed in Figure 6.6. On the one hand, this is the distance to surrounding content (CSS properties `margin` or `left-/top-/right-/bottom-margin`, resp.). Furthermore, the distance of the box's border to any contained content can be defined (properties `padding` or `left-/top-/right-/bottom-padding`, resp.). Finally, the width of a box's border is defined by the properties `border` or `left-/top-/right-/bottom-border`, resp. We will later come back to these settings.

Boxes can be positioned in a variety of possibilities. The positioning mode is set in the CSS property `position`, which can have one of the four values `static`, `absolute`, `fixed` and `relative`. The default setting is `static`, which does not affect the standard element flow (top to bottom on the web page). Boxes can furthermore be po-

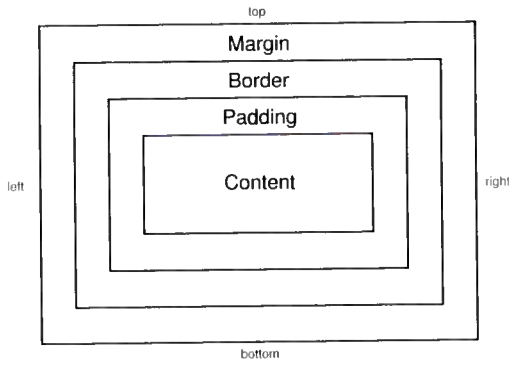

Figure 6.6: CSS Box Model (W3C, 2007); box dimensions

sitioned absolutely to either the HTML document under consideration (`absolute`) or the viewport, i.e. the browser window or a page in print (`fixed`). Finally, boxes can be positioned relative to each other, using the setting `relative`.

An absolutely positioned box is provided with up to four additional parameters determining its position: a `left`, `right`, `top` and `bottom` offset. A box, which is positioned w.r.t. the upper left corner of the HTML document can, for instance, be determined by a `left` and a `top` offset in addition to its `width` and `height`. A box with the lower right corner as its fix point would accordingly be defined using the `bottom` and `right` offset parameters and leaving the other ones undefined. For boxes, which are overdefined, e.g. by defining a `left` and a `right` offset as well as a `width`, the W3C documents define the correct handling.

While the position of an absolutely positioned box is only determined by the given offsets, the position of a relatively positioned box must be computed regarding the other boxes on the same web page. One issue which determines the position of a relatively positioned `div`-box w.r.t. another one is its position in the DOM-tree of the HTML document. If a box A appears before another box B in the tree, then A is rendered above or left of B. If A appears after B, then it is rendered either right of B or below B. Furthermore, A can enclose B, if B is a child node of A in the DOM tree.

The absolute position of the box is then computed as the summation of the other boxes' measurements. Assume, the box under consideration is anchored to the upper left corner of the web page. Then, its top offset is the sum of all top and bottom margins, widths of the top and bottom borders and the heights of all boxes *above* the one under consideration. The box's left offset is computed as the sum of all left and right margins, widths of the left and right borders and the widths of all boxes *left* the one under consideration. The right and bottom offsets are left undefined. The padding of the one box, which surrounds all the mentioned boxes, is taken into account by assuming, that the top margin of the top-most box and the left margin of the left-most box is at least as wide as the padding of the surrounding box.

6.2.2 Objective of the Case Study

The objective of our case study is to apply BAiT to testing the implementation of the *Gecko* rendering engine. BAiT has originally been designed as a tool for the test of data-oriented reactive systems in general. In this section, we report on a feasibility study to validate the applicability of BAiT to HTML rendering engines. These systems (or system components, resp.) are not reactive systems in the original sense.

Reactive systems are based on events being sent forth and back between several systems. These events can be parameterized with data. A rendering engine works differently: It is document-centered, i.e. it receives a document and renders it. While sending the document to the rendering engine is still comparable to the reactive systems described above, rendering this document is not. It is no reaction in the original sense, since no evaluable events are sent back from the IUT. The only event sent back from the renderer states, that rendering is finished, but it does not contain the actual result of rendering. In many cases, the result of rendering must even be evaluated visually, while for some aspects, the relevant information can be retrieved from the rendering engine and can be computer-processed.

Such an aspect are the positions of `div`-boxes to which we will restrict the test. Rules for positioning such boxes are defined in the CSS Box Model. We will, however, not consider the full box model, but restrict to a fragment of it.

First of all, we concentrate on the settings `absolute` and `relative` with a binding to the top left corner of the web page for the possible positioning of boxes. Secondly, we consider empty boxes of an explicitly defined width and height only in order to keep the results of rendering predictable by the test oracle. Boxes filled with content may lead to overflowing content which results in a correction by the rendering engine based on information about the viewport dimensions and the used font dimensions. Treating those details in this feasibility study would not be purposeful and furthermore would have required arithmetic division operations, which would complicate the specification in µCRL. For this reason, we do not regard (overflowing) content in this case study.

Thirdly, we limit the possible scales used in the design definition of a `div`-box. Normally, the position and size of a `div`-box is determined by distances, which can be defined in a variety of scales. Some of those some are absolute (pixels, didot points, pico points, inches, millimeters and centimeters) and other are relative to either the actual font setting (scales em and ex) or the rest of the page layout (percentages or the `auto` setting). In this case study, we only consider absolute lengths of scale `px` (pixel) in order to avoid scale conversions.

Finally, we use a "flat" model in our case study rather than one, which resembles the whole nested structure of a web page. This means, that we regard only a distinct box *testbox* and its absolute position on the web page. When we add another box to the web page, then we recompute the position of *testbox* as it has changed due to the newly added other box. This means for instance, that, if we add another box above the regarded *testbox*, the top offset of *testbox* is recomputed as the summation of the previous top offset, the top and bottom margins of the new box, the top and bottom

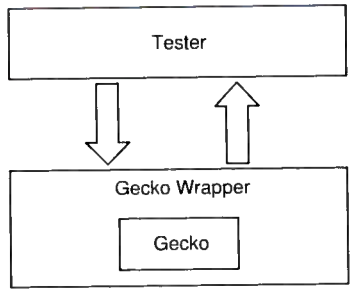

Figure 6.7: Test environment

border width of the new box and the new box's height. By doing so, we can easily keep track of the position of the regarded *testbox* without having to keep the whole HTML document structure in our model.

Apart from the applicability of BAiT to HTML rendering engines in particular, we also aim at two other targets with this case study. On the one hand, we want to test the adaptability of BAiT to nondeterministic behavior of the IUT further by introducing some artificial nondeterminism w.r.t. to the system's feedback about the rendered boxes. On the other hand, we want to regard the feasibility of μCRL as a language for the design of test purposes.

6.2.3 Realizing the Test Environment

In order to test Gecko, we first had to create a test environment. This environment, as schematically depicted in Figure 6.7, consists of a tester and an IUT. In our case, the tester is the tool BAiT. The IUT is a component named Gecko Wrapper, which wraps Gecko internally. Both the tester and the IUT are Java components which communicate with each other using bidirectional procedure-based communication.

In order to generate and run the tests, we also need a system specification of the CSS box model for Gecko and a test purpose to sketch out the later test cases. While the design of the tests and hence also that of the test purposes will be the topic of Section 6.2.5, we will in the remainder of this section discuss the specification of the boxmodel. Furthermore, we will give some details on the Gecko Wrapper.

6.2.4 Modelling CSS in μCRL

We modeled a fragment of the CSS box model with the limitations from Section 6.2.2 in μCRL. The modeled fragment of CSS allows to position boxes relative to each other or absolutely. In our model, recursive structures of boxes are flattened by regarding only one distinct box and its position, rather than a structure of boxes. Whenever a box is added, only the consequences on the position of the regarded box are computed and applied.

Action	Functionality
Input actions:	
`resetBoxes`	Wipes all boxes and starts again with a fresh document.
`setupTestbox`	Defines the distinctly regarded test box.
`putBoxRelative`	Puts a box relative to the other boxes yet defined in the actual HTML document. It can be defined, whether this box appears left of, right of, above, beneath or around all yet defined boxes. Finally, the measurements can be defined.
`putBoxAbsolute`	Puts a box with an absolute position.
`render`	Renders the actually defined document and starts returning results (offsets, see below).
Output actions:	
`offsetLeft`	Returns absolute left offset of test box.
`offsetRight`	Returns absolute right offset of test box.
`offsetTop`	Returns absolute top offset of test box.
`offsetBottom`	Returns absolute bottom offset of test box.

Table 6.1: Actions for the CSS box model

While rendering a web page is in principle a document-centered task, our specification of the box model is behavior-oriented. Hence, we defined a set of input actions, which allow us to put boxes into a box structure. Furthermore, we defined some output actions, which provide information about the current offset of the regarded box to the tester. The actions are defined in Table 6.1.

The system behavior for the CSS box model is specified as follows: As a first step, a testbox must be set up (`setupTestbox`). The action `setupTestbox` accepts parameters, which determine the box's width and height. Other boxes can be put in the vicinity of this testbox using the actions `putBoxRelative` and `putBoxAbsolute` in any order. The action `putBoxRelative` accepts 15 parameters: The first one determines, whether the box appears above, below, left, right or around the other boxes, which have yet been inserted into the web page. This parameter is named "position", but is actually not related to the `position`-property of CSS. The next two parameters determine the box's width and height. The last 12 parameters, finally, define the width of the box's padding, border and margin as depicted in Figure 6.6. The action `putBoxAbsolute` only accepts seven parameters. The first three are identical with those from `putBoxRelative`, while the remaining four parameters define the box's absolute position on the page w.r.t. the four margins of the web page.

Any of these three actions can be followed by an action `resetBoxes` in order to delete all boxes and start from scratch, or by an action `render`. In this case, the IUT renders the defined structure of `div`-boxes. Afterwards, the different actual values for the offsets of the testbox (left, right, top, bottom offset) are returned by the IUT in an arbitrary order.

As we described in Sect. 6.2.2, we only regard a distinct box *testbox* in the model, whose position we recompute each time another box is added to the HTML test document. The actions `putBoxAbsolute` and `putBoxRelative` change this position in the described way. In Figure 6.8, we give the μCRL code, which relates to the behavior of `putBoxRelative`.

The fragment of our specification shows the definition of the action `putBoxRelative` within a process `PrepareRendering`. When `putBoxRelative` has been invoked, the system enters another process, `PositionBoxRelative`. After a τ-step, the system leaves `PositionBoxRelative` and goes back to `PrepareRendering`. While doing so, the new position of the test box is computed depending on the value of the variable `pposition` of the newly positioned box. This leads to a case distinction depending on the position of the new box.

Wrapping Mozilla Gecko

Mozilla Gecko can be embedded into custom applications as a component, which can be programmed using its XPCOM interfaces. The Cross Platform Component Object Model (XPCOM) is an unmanaged component framework, which is used for the Mozilla software products (Turner and Oeschger, 2003). Gecko can be embedded into Java applications. In order to do so, one can either instantiate it directly via its XPCOM interfaces or embed it indirectly via the `Browser` component of the Standard Widget Toolkit (SWT)[6].

We implemented a wrapper for Gecko in Java using SWT. The wrapper receives all actions which place boxes and builds up an internal structure for a test web page. On action `render`, the wrapper generates actual HTML and CSS code and sends this code to the renderer. A window is opened in which rendering takes place.

When rendering is finished, the renderer is queried for the offsets. For this procedure, we followed an existing code example[7]. In order to query an offset, a short piece of JavaScript code is generated and executed within the web browsing component. This piece of code internally queries for the respective offsets and writes the result to an (invisible) status bar. When this has happened, the wrapper can read the value from this status bar in order to store it, and the next piece of JavaScript is generated and executed (one execution per one of the four offset parameters). After all offsets have been queried, the tester is informed by the actions `offsetLeft`, `offsetRight`, `offsetTop` and `offsetBottom` in a random order. We chose for a random order in order to test the adaptation of BAiT to nondeterministic behavior of the IUT, as we have described in Section 6.2.2.

[6] `http://www.eclipse.org/swt`

[7] `http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/org.eclipse.swt.snippets/src/org/eclipse/swt/snippets/Snippet160.java`

$$
\begin{aligned}
&\ldots\\
&\mathrm{PrepareRendering}(\mathrm{position} : \mathrm{PositionType}, \mathrm{relation} : \mathrm{RelationType},\\
&\ \mathrm{width} : \mathrm{Nat}, \mathrm{height} : \mathrm{Nat}, \mathrm{offsetLeft} : \mathrm{Nat}, \mathrm{offsetRight} : \mathrm{Nat},\\
&\ \mathrm{offsetTop} : \mathrm{Nat}, \mathrm{offsetBottom} : \mathrm{Nat}) =\\
&\ldots\\
&+ \sum_{\mathrm{pposition:PositionType}} \sum_{\mathrm{pwidth:Nat}} \sum_{\mathrm{pheight:Nat}} \ldots\\
&\quad \mathrm{putBoxRelative}(\mathrm{pposition}, \mathrm{pwidth}, \mathrm{pheight}, \mathrm{pmarginleft},\\
&\qquad \mathrm{pmarginright}, \mathrm{pmargintop}, \mathrm{pmarginbottom}, \mathrm{pborderleft}, \ldots).\\
&\quad \mathrm{PositionBoxRelative}(\mathrm{position}, \mathrm{relation}, \mathrm{width}, \mathrm{height}, \ldots) \ldots\\
&\ldots\\
&\mathrm{PositionBoxRelative}(\mathrm{position} : \mathrm{PositionType}, \mathrm{relation} : \mathrm{RelationType},\\
&\ \mathrm{width} : \mathrm{Nat}, \mathrm{height} : \mathrm{Nat}, \ldots) =\\
&\ \tau.\mathrm{PrepareRendering}(\ldots, \mathrm{offsetLeft} + \mathrm{pmarginLeft} + \mathrm{pborderLeft} +\\
&\qquad \mathrm{ppaddingLeft} + \mathrm{pwidth} + \mathrm{ppaddingRight} + \mathrm{pborderRight} +\\
&\qquad\qquad \mathrm{pmarginRight}, \ldots)\\
&\qquad \triangleleft \mathrm{and}(\mathrm{eq}(\mathrm{relation}, \mathrm{relative}), \mathrm{eq}(\mathrm{pposition}, \mathrm{left})) \triangleright \delta\\
&+ \tau.\mathrm{PrepareRendering}(\ldots, \mathrm{offsetLeft}, 0, \mathrm{offsetTop}, \mathrm{offsetBottom})\\
&\qquad \triangleleft \mathrm{and}(\mathrm{eq}(\mathrm{relation}, \mathrm{relative}), \mathrm{eq}(\mathrm{pposition}, \mathrm{right})) \triangleright \delta\\
&+ \tau.\mathrm{PrepareRendering}(\ldots, \mathrm{offsetTop} + \mathrm{pmarginTop} + \mathrm{pborderTop} +\\
&\qquad\qquad \mathrm{ppaddingTop} + \mathrm{pheight} + \mathrm{ppaddingBottom} +\\
&\qquad\quad \mathrm{pborderBottom} + \mathrm{pmarginBottom}, \mathrm{offsetBottom})\\
&\qquad \triangleleft \mathrm{and}(\mathrm{eq}(\mathrm{relation}, \mathrm{relative}), \mathrm{eq}(\mathrm{pposition}, \mathrm{top})) \triangleright \delta\\
&+ \tau.\mathrm{PrepareRendering}(\ldots, \mathrm{offsetLeft}, \mathrm{offsetRight}, \mathrm{offsetTop}, 0)\\
&\qquad \triangleleft \mathrm{and}(\mathrm{eq}(\mathrm{relation}, \mathrm{relative}), \mathrm{eq}(\mathrm{pposition}, \mathrm{bottom})) \triangleright \delta\\
&\ldots
\end{aligned}
$$

Figure 6.8: Excerpt from the CSS box model fragment in μCRL

6.2.5 Running the Tests

Design of the Test Cases

In the BAiT approach, test generation is based on enumerative test case generation, so we applied data abstraction on the specification of the system, in order to avoid space explosion induced by the many unrestricted numerical parameters of the input actions `setupTestbox`, `putBoxAbsolute` and `putBoxRelative`.

The second step was to design test purposes. We designed two test purposes, of which one traditionally directly as an LTS, while the second one was specified in μCRL as was the system itself. According to the first test purpose, we set up a testbox, put at least one more (relatively positioned) box in its vicinity and render the resulting HTML document. Having done so, we expect the system to return at least the top offset of the testbox. The second test purpose is designed a bit differently, since we still wanted to experiment more with BAiT's capability of behavior-adaptation during a test run. For this purpose, the test purpose was designed to expect at least the left offset of the testbox and to refuse an action `offsetBottom` following directly on the `render` action. This refusal in combination with the absolutely random order of `offset`-events from the IUT leads to more situations in which BAiT will be led into a trace to an Inconc verdict, from which it will try to find an alternative trace to a Pass verdict. It will, however, never find such a trace and it will have to give up, terminating with verdict Inconc. Since the generated test cases can contain loops, BAiT might search for a trace to Pass without ever terminating. This issue has been solved by introducing a configuration option for BAiT, which defines the maximum amount of traces to search for before giving up and assigning Inconc. This second test purpose is shown in Figure 6.9 as a μCRL specification and an LTS.

In a third step, we generated CTGs with TGV. The abstracted system specification as input to TGV was quite manageable with its 17 states and 57 transitions, so that the generation process took place within negligible time. For the first test purpose, generation resulted in a CTG with 25 states and 70 transitions. The second test purpose put more restrictions on the behavior of the IUT during the test, so the number of transitions in the resulting CTG was reduced to 59; the number of states increased slightly to 28. In general, these numbers are relatively low, a circumstance which does not astonish if one keeps in mind, that we regard only the behavior of a highly data-intensive system after data abstraction. The main work, as we had already remarked earlier, is the data selection during test execution.

Test Execution

Based on the generated CTGs, we ran some tests with BAiT and the Gecko wrapper. We used the default trace search algorithm of BAiT in order to select test traces through the CTGs. This algorithm searches only for traces to Pass, using a breadth-first search. Having automatically selected a trace to Pass, we then selected data for the different parameters of the box positioning actions and executed the trace.

Test purpose in μCRL:

```
proc  PASS    =  ACCEPT.PASS
      FAIL    =  REFUSE.FAIL
      PutBox  =  render + putBoxRelative.PutBox
      TP      =  setupTestbox.putBoxRelative.PutBox.
                 (offsetLeft.PASS + offsetBottom.FAIL)

init  TP
```

Resulting test purpose as LTS before adding placeholders for action parameters:

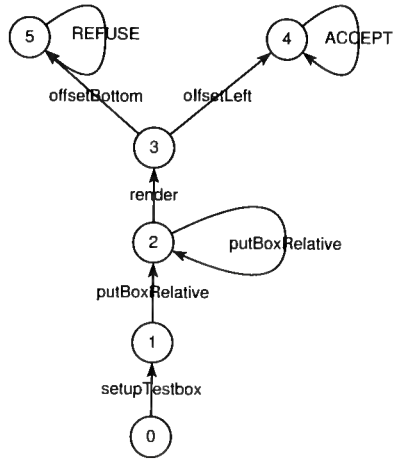

Figure 6.9: A test purpose both in μCRL and as an LTS

During the different test runs, we found a few failures. However, those failures were induced by faults in the used model rather than by the IUT itself. After having eliminated the faults, we did not find any more failures in the IUT.

As expected, the test runs based on the first CTG always ended in a Pass verdict, after we had corrected the model. The test runs based on the second CTG, randomly went to a Pass or an Inconc verdict. This behavior was dependent on whether the wrapper returned an `offsetBottom` event before (Inconc) or after (Pass) the `offsetLeft` event (cf. the description of the second test purpose). Since the order of events was implemented in the wrapper to be random, the assignment of verdicts was also as expected a priori.

Chapter 7

Bug Hunting with False Negatives

> Sometimes you're the windshield,
> sometimes you're the bug.
>
> *(Dire Straits)*

We had discussed the application of testing to software quality assurance in the previous chapters. While model-based conformance testing is a rigorous and well-established approach to find faults in a software system, it has, however, one major deficiency: It tests whether an implementation of a model conforms to this model, no matter, whether the model itself is correct.

A correct software specification or model accurately resembles the requirements one has with respect to the modeled software product. In order to verify this, the technique of model checking (Clarke et al., 1999; Bérard et al., 2001) has been developed. Model checking takes as input a formal model, like a transition system, and a logical formula, which defines the requirement under consideration. Then, the model is checked whether it satisfies the requirement, and the model checker returns a counterexample, if this is not the case. At the hand of this trace, the responsible bug can be traced back.

Model Checking and Abstractions

As we could already see from the previous chapters, abstractions are widely used to reduce the state space of complex, distributed, data-oriented and thus large systems for verification purposes. The application of abstraction techniques to such systems also has a great impact on the applicability of model-checking techniques on them. In this chapter, we want to develop an approach for the verification of data-oriented systems using temporal logic to express the verified requirements.

In our approach, we focus on abstractions that are used to check satisfaction rather than the violation of properties. These abstractions are constructed in such a way that we can transfer positive verification results from the abstract to the concrete model, but not the negative ones. Since we discuss in this chapter, how to show the *correctness* of a system, a positive result means that the system is correct, while a negative result means that it is not. Positive and negative have thus the exactly reversed meaning than in the previous, testing-related, chapters. The advantage of our approach is, that we do not exclude possible bugs from the very beginning of system validation.

However, counterexamples found on the abstract system may have no counterpart in the concrete system. This problem had already been discussed in Chapter 4 in the

context of software test generation. We will further refer to this kind of counterexamples as *false negatives*. Usually, as has been discussed by Lakhnech et al. (2001) Das and Dill (2002) and Clarke et al. (2003), false negatives are used to refine the abstraction and iteratively call the model checking algorithm on the refined abstraction.

In this chapter, we consider false negatives in the context of data abstractions. As an illustrating example, we use the timer abstraction from Dams and Gerth (1999). In the course of this abstraction, a certain value k is defined, below which all timer values are left unchanged, while all values greater than k are mapped to an abstract value k^+. In doing so, the deterministic time progress operation $\mathtt{tick}$ (decreasing the values of active timers by one), becomes nondeterministic in the abstract model, as can be seen in Figure 7.1. The advantage of this abstraction is, however, that we only have to regard the k smallest values and the constant k^+ in order to prove that a property holds for any value n.

Example 7.1. Consider a system, where for some constant value n, every timer setting $\mathtt{set}(n)$ is followed by n $\mathtt{tick}$-steps before the timer is set again. Being set to a value $n > k$, the abstract timer can do an arbitrary number of $\mathtt{tick}$-steps, before it reaches the value $k-1$. Only from there it decreases until it expires at 0.

We now use this timer abstraction to verify an action-based LTL property $\Box(a \rightarrow \Diamond b)$ and obtain the following trace as a counterexample for the abstract system: $a.\mathtt{set}(k^+).\mathtt{tick}^3.b.(a.\mathtt{set}(k^+).\mathtt{tick}^2.d)^*$.

The timer abstraction has obviously affected the parameter of the $\mathtt{set}$ action, so that the number of $\mathtt{tick}$-steps following $\mathtt{set}(k^+)$ is not fixed anymore. As a result, this trace is a *false negative*, since it does not reflect any possible trace of the original system (remember the constant n).

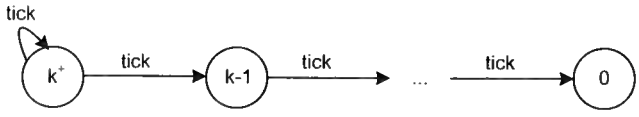

Figure 7.1: Abstracted timer

Assuming that the trace $a.\mathtt{set}(n).\mathtt{tick}^n.b.(a.\mathtt{set}(n).\mathtt{tick}^n.d)^*$ exists in the original system, the false negative still contains a clue for finding this concrete counterexample. We can relax the found abstract counterexample by using the information that the operations on timers are influenced by the timer abstraction and check whether the concrete system contains a trace matching the pattern $a.\mathtt{any}^*.b.(a.\mathtt{any}^*.d)^*$, where $\mathtt{any}$ represents any action on timers. We call such a pattern a *violation pattern*. Note that any trace matching the violation pattern violates our property of interest. The pattern contains a cyclic part, and it is more restrictive than the negation of the property. Therefore, when enumerative model checking is concerned, it is easier to find a trace of the concrete system satisfying the pattern than one that violates the property.

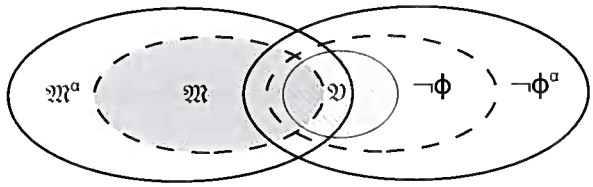

Figure 7.2: Violation pattern approach

In this chapter, we propose a framework developed in Calamé et al. (2007b) that supports the bug hunting process described in the above example. In this framework, we apply a combination of abstraction, refinement and constraint solving techniques to process algebraic specifications. The framework is illustrated in Figure 7.2 where $\mathfrak{M}$ denotes the concrete system, $\mathfrak{M}^\alpha$ stands for an abstraction of $\mathfrak{M}$, ϕ is the property in question and ϕ^α is its abstraction. When checking whether the abstract system satisfies the abstract property, we may obtain a counterexample having no counterpart in the concrete system (the set $(\mathfrak{M}^\alpha \backslash \mathfrak{M}) \cap \neg\phi$). Given the counterexample, we relax actions influenced by the data abstraction and construct a violation pattern that represents a set of traces violating the property and resembling the counterexample. For this to work, we need an accurate analysis of contracting and precise abstractions as developed by Kesten and Pnueli (2000). In short, contracting abstractions abstract a system property in a way, that less traces fulfill this property, while precise abstractions do not affect fulfilling traces.

To check whether there is a concrete trace matching the violation pattern, we transform the violation pattern and the specification of the concrete system into a constraint logic program. Subsequently, a constraint solver is used to find a concrete trace matching the violation pattern, if such a trace exists.

The rest of the chapter is organized as follows: In Section 7.1, we give a brief overview on LTL. In Section 7.2, we define a next-free action-based LTL and extend it by data. In Section 7.3, we work out abstractions of LTSs and of eALTL properties. In Section 7.4, we present a taxonomy of counterexamples, of which we select the false negatives to build up a bug hunting framework and discuss its correctness in Section 7.5. In Section 7.6, we give an example for the implementation of this framework. Finally, we discuss related work in Section 7.7.

7.1 Linear Temporal Logic

Requirements to a system, also named system properties, are formally noted as logical formulae in a temporal logic. A temporal logic extends static logic ("It is raining." $\rightarrow \{\top, \bot\}$) by the dimension of time, so that the status of the described objects can change over time. Such a temporal formula, like "The sun is shining, but at some point in future, it will be raining." can be evaluated to a static value from the range

$\{\top, \bot\}$, i.e. even though the status of the described objects changed dynamically, the value of the formula does not.

There exist several theories for temporal logic, of which the one discussed in this thesis is the Linear Temporal Logic (LTL) as it has, for instance, been described in Clarke et al. (1999). In LTL, formulae over traces in a system can be formulated, which state requirements from *states* of the system. Besides the standard operators for first-order logic, LTL also provides operators which describe the development of a requirement on the temporal dimension of the trace under consideration. In particular, those operators allow to formulate

- what is the requirement for the *next* state,
- what is the requirement for a state *sometime in the future*,
- what should *always* hold on a trace, and
- what should hold *until* something else holds.

The latter operator even exists in a stronger and a weaker form, either requiring the second statement to hold at some point in future, or not. While in other theories, temporal statements can be made about sets of traces within a system, LTL formulae can only be formulated for *all* traces. Finally, there also exists a subtheory LTL_{-X}, which defines an LTL without the *next*-operator, the first item in our list. In the remainder of this chapter, we will address a *next*-free theory.

7.2 Action-based LTL and Data

The classical temporal logics like Linear Temporal Logic (LTL) are state-based. This means that they consider the states of a system as the objects under consideration, not the transitions between these states. This perception is not always optimal, and so Giannakopoulou (1999) developed an action-based variant, Action-based Linear Temporal Logic (ALTL), which considers transitions rather than states. In this chapter, we propose a data extension for ALTL, eALTL. This logic specifies system properties in terms of events parameterized with data. Here, we first define *action formulae*, their satisfaction, and then define eALTL.

With an action formula, statements can be made on the level of single action labels in an LTS using the standard boolean operations for $\top$, negation, conjunction, and disjunction. Furthermore, the statement can be made, whether an action satisfies requirements regarding the action's name and its parameters. The latter is expressed as the membership in a particular set.

Definition 7.2 (Action Formulae). Let x be a variable from Var, expr be a boolean expression from Exprs and a be an event from Events. The syntax of an action formula ζ is defined as follows:

$$\zeta ::= \top \mid \{a(x) \mid \mathit{expr}(x)\} \mid \neg\zeta \mid \zeta \wedge \zeta \mid \zeta \vee \zeta$$

■

We will use $a(x)$ as an abbreviation for $\{a(x) \mid \top\}$ and $a(d)$ as an abbreviation for $\{a(x) \mid x = d\}$. We do not impose any limitations on the set of boolean expressions.

Definition 7.3 (Interpretation of an action formula). Let $\lambda \in \Lambda$ and ζ be an action formula. The satisfaction of ζ on λ is defined as follows:

λ	$\models$	$\top$	always
λ	$\models$	$\{a(x) \mid expr(x)\}$	if there exists some $d \in \mathbb{D}$ s.t. $\lambda = a(d)$ and $[\![expr]\!]_{[x \mapsto d]} = \top$
λ	$\models$	$\neg\zeta$	if not $\lambda \models \zeta$
λ	$\models$	$\zeta_1 \wedge \zeta_2$	if $\lambda \models \zeta_1$ and $\lambda \models \zeta_2$
λ	$\models$	$\zeta_1 \vee \zeta_2$	if $\lambda \models \zeta_1$ or $\lambda \models \zeta_2$

■

The next two definitions extend the scope of action formulae to traces. In order to do so, we introduce the notion of eALTL formulae, which allow, besides action formulae and the standard boolean operations, to express the temporal dimension of a requirement to a system. In particular, we introduce the operators $\Box$ for requirements on *every* action of a trace, $\Diamond$ for requirements on an action somewhere in the *future* of the current trace, and the two operators **U** (until) and **R** (release). The latter two operators regard two properties of which one holds until the second one holds. With **U**, the second property must hold at some point, with **R**, this second property holds optionally.

Definition 7.4 (eALTL Formulae). Let ζ be an action formula. The syntax of eALTL formulae is defined by the following grammar:

$$\phi ::= \zeta \mid \neg\phi \mid \Box\phi \mid \Diamond\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \mathbf{U} \phi \mid \phi \mathbf{R} \phi$$

■

Remark 7.5. As we will see later in this section, the theory would also work without $\Box$, $\Diamond$ and **R** being explicitly defined. However, we define them here for reasons of convenience.

Definition 7.6 (Semantics of eALTL). Let π be an infinite trace, ϕ, ψ be eALTL formulae and ζ be an action formula. Then:

π	$\models$	ζ	if $\pi[1] \models \zeta$
π	$\models$	$\neg\phi$	if not $\pi \models \phi$
π	$\models$	$\Box\phi$	if $\forall i \in \mathbb{N} \setminus \{0\} : \pi^i \models \phi$
π	$\models$	$\Diamond\phi$	if $\exists i \in \mathbb{N} \setminus \{0\} : \pi^i \models \phi$
π	$\models$	$\phi \wedge \psi$	if $\pi \models \phi$ and $\pi \models \psi$
π	$\models$	$\phi \vee \psi$	if $\pi \models \phi$ or $\pi \models \psi$
π	$\models$	$\phi \mathbf{U} \psi$	if there exists $k \in \mathbb{N} \setminus \{0\}$ such that for all $0 < i < k : \pi^i \models \phi$ and $\pi^k \models \psi$
π	$\models$	$\phi \mathbf{R} \psi$	if if (sic!) for any $i \in \mathbb{N} \setminus \{0\} \pi^i \not\models \psi$, then $\pi^i \models \phi$

■

Let $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{init})$ be an LTS. We say that $\mathfrak{M} \models \phi$ if and only if $\pi \models \phi$ for all traces π of $\mathfrak{M}$ starting at σ_{init}. We can identify the following equivalences:

$$\bot \equiv \neg\top; \tag{7.1}$$
$$\Diamond\phi \equiv \top\mathbf{U}\phi; \tag{7.2}$$
$$\Box\phi \equiv \neg\Diamond\neg\phi; \tag{7.3}$$
$$\phi \Rightarrow \psi \equiv \neg\phi \vee \psi; \tag{7.4}$$
$$\phi\mathbf{R}\psi \equiv \neg(\neg\phi\mathbf{U}\neg\psi). \tag{7.5}$$

eALTL is suitable to express a broad range of property patterns like occurrence, bounded response or absence (Dwyer et al., 1999).

7.3 Abstracting eALTL

In this section, we present an abstraction mechanism based on homomorphisms as in Clarke et al. (1994) and Kesten and Pnueli (2000), and adapted to an action-based setting. Abstracting a system leads to a smaller state space, which can thus be examined easier. However, model checking an abstracted system also requires the abstraction of the properties that have to be checked. We will first present the abstraction of systems and then the abstraction of eALTL properties.

7.3.1 Abstraction of a system

The basis for the abstraction is a homomorphism $\alpha = \langle h_s, h_a \rangle$ defining two abstraction functions which regard states and actions of an LTS (Van de Pol and Valero Espada, 2004; Clarke et al., 1994). The function $h_s : \Sigma \to \Sigma^\alpha$ maps the states of a concrete system $\mathfrak{M}$ to abstract states. The function $h_a : \Lambda \to \Lambda^\alpha$ does the same with the action labels of $\mathfrak{M}$. Abstracting a system using homomorphisms has been introduced in Section 2.5.

Definition 7.7 (Abstraction Homomorphism). Let abstraction $\alpha = \langle h_s, h_a \rangle$ for automaton $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{init})$ be given. We define $\alpha(\mathfrak{M})$ to be $(\Sigma^\alpha, \Lambda^\alpha, \Delta^\alpha, h_s(\sigma_{init}))$, where $\sigma^\alpha \xrightarrow{\lambda^\alpha} \hat{\sigma}^\alpha \in \Delta^\alpha$ if and only if $\sigma \xrightarrow{\lambda} \hat{\sigma} \in \Delta$, for some σ, $\hat{\sigma}$ and λ such that $h_s(\sigma) = \sigma^\alpha$, $h_s(\hat{\sigma}) = \hat{\sigma}^\alpha$, and $h_a(\lambda) = \lambda^\alpha$. ■

Now, we have to consider trace inclusion again. In order to preserve all behavior of the original system in the abstract system, we have to make sure, that there are abstract counterparts for all traces of the original system.

Definition 7.8 (Trace Inclusion w.r.t. α). Let $\alpha = \langle h_s, h_a \rangle$ be a homomorphism. Assuming a trace $\pi \in [\![\mathfrak{M}]\!]_{traces}$, we define $\pi^\alpha = h_a(\pi)$ with $h_a(\pi)[i] = h_a(\pi[i])$ for all $i \in \mathbb{N} \setminus \{0\}$.

We say that $\mathfrak{M} \subseteq_\alpha \mathfrak{M}^\alpha$ if and only if for every trace π of $\mathfrak{M}$ there exists a trace $\alpha(\pi) \in [\![\mathfrak{M}^\alpha]\!]_{traces}$. ■

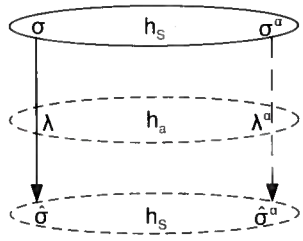

Figure 7.3: Abstraction requirement for LTSs

It is well known that homomorphic abstractions lead to overapproximations. In particular, the abstract system covers at least the traces of the concrete system.

Lemma 7.9. Let $\mathfrak{M}$ be an LTS with homomorphism α. Then $\mathfrak{M} \subseteq_\alpha \alpha(\mathfrak{M})$. ■

Proof. Assume an LTS $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{\text{init}})$ and an arbitrary trace $\pi \in [\![\mathfrak{M}]\!]_{\text{traces}}$ with its state projection π_σ. This means that $\forall i \in \mathbb{N} : \pi_\sigma[i] \xrightarrow{\pi_\Lambda[i+1]} \pi_\sigma[i+1] \in \Delta$.

Let us now define an automaton $\alpha(\mathfrak{M}) = (\Sigma^\alpha, \Lambda^\alpha, \Delta^\alpha, \sigma^\alpha_{\text{init}})$ and an abstract trace $\pi^\alpha = \alpha(\pi)$ following Definition 7.8. We now have to prove, that $\pi^\alpha \in [\![\alpha(\mathfrak{M})]\!]_{\text{traces}}$:

1. In a first step, we have to prove, that $\pi^\alpha_\sigma = h_s(\sigma_{\text{init}})$. Traces always start in the initial state of their LTS, so we can safely claim that $\pi_\sigma[0] = \sigma_{\text{init}}$. We have defined π^α so, that $\forall i \in \mathbb{N} : \pi^\alpha_\sigma = h_s(\pi_\sigma[i])$. For $i = 0$, this automatically means that $\pi^\alpha_\sigma[0] = h_s(\pi_\sigma[0]) = h_s(\sigma_{\text{init}})$.

2. For an arbitrary $i \in \mathbb{N}$, we have $\pi_\sigma[i] \xrightarrow{\pi_\Lambda[i+1]} \pi_\sigma[i+1] \in \Delta$. By Definition 7.7, we have $h_s(\pi_\sigma[i]) \xrightarrow{h_a(\pi_\Lambda[i+1])} h_s(\pi_\sigma[i+1]) \in \Delta^\alpha$. So for any $i \in \mathbb{N}$, we have $\pi^\alpha_\sigma[i] \xrightarrow{\pi^\alpha_\Lambda[i+1]} \pi^\alpha_\sigma[i+1] \in \Delta^\alpha$.

From the above, we may conclude, that for an arbitrary trace $\pi \in [\![\mathfrak{M}]\!]_{\text{traces}}$ there exists a trace $\alpha(\pi) \in [\![\alpha(\mathfrak{M})]\!]_{\text{traces}}$ and that thus $\mathfrak{M} \subseteq_\alpha \alpha(\mathfrak{M})$. □

It is often more convenient to apply abstractions directly on a system specification $\mathfrak{S}$ than on its transition system $\mathfrak{M}$. Such an abstraction on the level of $\mathfrak{S}$ is well-developed within the *Abstract Interpretation* framework (Cousot and Cousot, 1977; Dams, 1996; Dams et al., 1997). Abstract Interpretation imposes a requirement on the relation between the concrete specification $\mathfrak{S}$ and its abstract interpretation $\mathfrak{S}^\alpha$. This takes the form of a safety requirement on the relation between data and operations of the concrete system and their abstract counterparts (we skip the details). Each value of the concrete domain $\mathbb{D}$ is related by a data abstraction function h_d to a value from the abstract domain $\mathbb{D}^\alpha$. For every operation (function) f on the concrete data domain, an abstract function f^α is defined, which overapproximates f. For reasons of simplicity, we assume f to be a unary operation. Furthermore, we apply only data abstraction. This means that the names of actions in a system are not affected by the

abstraction, i.e. $h_a(a(d)) = a(h_d(d))$, so that two actions $a(x)$ and $b(y)$ cannot be mapped to the same abstract action.

However, applying abstractions directly on a system's specification $\mathfrak{S}$ rather than on its LTS leads to a loss of precision. Let $\mathfrak{S}^\alpha$ be the abstract interpretation of $\mathfrak{S}$, and let $\mathfrak{M}^\alpha$ and $\mathfrak{M}$ be their underlying LTSs. It is well known that $\mathfrak{M}^\alpha$ is only an overapproximation of $\alpha(\mathfrak{M})$, with $\alpha(\mathfrak{M})$ denoting the abstraction of $\mathfrak{M}$ on the level of LTSs here (cf. Clarke et al., 1994). In particular, we will still have trace inclusion up to α: $\mathfrak{M} \subseteq_\alpha \alpha(\mathfrak{M}) \subseteq_\alpha \mathfrak{M}^\alpha$.

7.3.2 Abstraction of eALTL formulae

The abstraction of eALTL formulae is based on the notions of *contracting and precise abstractions* as introduced by Kesten and Pnueli (2000). In a contracting abstraction, a property ϕ^α holds for a trace π^α if and only if the property ϕ holds for *all* concrete traces π with $\pi^\alpha = \alpha(\pi)$. Note that for soundness of abstract model checking, we need contracting abstractions. This does, however, not imply that all properties that hold for the original system, *must* also hold in the abstract system (see Figure 7.4, ellipse vs. the hatched square). In *precise* abstractions, this cannot happen.

Definition 7.10 (Contracting and Precise Abstraction). Let ϕ be a property over an alphabet Λ. Its abstraction ϕ^α is

contracting if and only if: $\forall \pi \in \Lambda^* : \alpha(\pi) \models \phi^\alpha \Rightarrow \pi \models \phi$.

precise if and only if: $\forall \pi \in \Lambda^* : \alpha(\pi) \models \phi^\alpha \Leftrightarrow \pi \models \phi$.

■

In the following, we will define an abstraction of eALTL formulae that is guaranteed to be contracting. We will first consider action formulae. For the standard boolean operations, as well as for $\top$ and $\bot$, abstractions are straight forward. The difficult part are those formulae, where statements are made whether an action label belongs to a particular set of labels or not. We abstract those as follows: In the positive case, we preserve the name of the action under consideration, since we only abstract data here. For parameters, we check whether all concrete data values for the abstract one fulfill the original data property $expr(x)$. For the abstraction of the negation of such a set formula, we require either the action name to be different or none of the concrete data values for the abstracted parameters to fulfill the expression. This is not exactly the inverse of the abstracted positive set formula, but we want to achieve a contracting abstraction as shown in Figure 7.4. Doing so is safer w.r.t. counterexamples found for the abstract system, since the abstract system does not fulfill more properties than the original one.

Definition 7.11 (Abstraction of Action Formulae). Action formulae as defined in Definition 7.2 are abstracted as follows:

$$\alpha(\top) := \top \tag{7.6}$$
$$\alpha(\bot) := \bot \tag{7.7}$$
$$\alpha(\{a(x) \mid expr(x)\}) := \{a(x^\alpha) \mid \forall x : h_d(x) = x^\alpha \rightarrow expr(x))\} \tag{7.8}$$
$$\alpha(\neg\{a(x) \mid expr(x)\}) := \bigvee_{b \neq a} \{b(x^\alpha)\} \vee \{a(x^\alpha) \mid \forall x : h_d(x) = x^\alpha \rightarrow \neg expr(x)\} \tag{7.9}$$
$$\alpha(\zeta_1 \wedge \zeta_2) := \alpha(\zeta_1) \wedge \alpha(\zeta_2) \tag{7.10}$$
$$\alpha(\zeta_1 \vee \zeta_2) := \alpha(\zeta_1) \vee \alpha(\zeta_2) \tag{7.11}$$

■

Remark 7.12. It will be silently assumed that, due to the laws of De Morgan (1860), the definition of α can be extended as follows:

$$\alpha(\neg(\zeta_1 \wedge \zeta_2)) \equiv \alpha(\neg\zeta_1 \vee \neg\zeta_2) \equiv \alpha(\neg\zeta_1) \vee \alpha(\neg\zeta_2) \tag{7.12}$$
$$\alpha(\neg(\zeta_1 \vee \zeta_2)) \equiv \alpha(\neg\zeta_1 \wedge \neg\zeta_2) \equiv \alpha(\neg\zeta_1) \wedge \alpha(\neg\zeta_2) \tag{7.13}$$

■

Now, we define the abstraction of eALTL formulae for traces. The basic principle is to inductively trace the abstraction of eALTL formulae back to that of action formulae. For negated eALTL formulae, however, this is not possible, since the abstraction would then not be contracting anymore. As can be seen in Figure 7.4, the abstraction of a negated formula $\alpha(\neg\phi)$ is not equal to the negation of an abstracted formula $\neg\alpha(\phi)$. In order to preserve the contraction of our abstraction, we have to transform negated formulae first into a form, where the negation operator is as innermost as possible, before we abstract any further. The transformation itself is mainly based on the equivalences stated in Section 7.2 (and here especially equations (7.3) and (7.5)) as well as the laws of De Morgan from the previous remark.

Definition 7.13 (Abstraction of eALTL Formulae). eALTL formulae as defined in Definition 7.4 are abstracted as follows:

$$\alpha(\Box\phi) := \Box\alpha(\phi) \tag{7.14}$$
$$\alpha(\neg\Box\phi) := \Diamond\alpha(\neg\phi) \tag{7.15}$$
$$\alpha(\Diamond\phi) := \Diamond\alpha(\phi) \tag{7.16}$$
$$\alpha(\neg\Diamond\phi) := \Box\alpha(\neg\phi) \tag{7.17}$$
$$\alpha(\phi \wedge \psi) := \alpha(\phi) \wedge \alpha(\psi) \tag{7.18}$$
$$\alpha(\phi \vee \psi) := \alpha(\phi) \vee \alpha(\psi) \tag{7.19}$$
$$\alpha(\phi \mathbf{U} \psi) := \alpha(\phi)\mathbf{U}\alpha(\psi) \tag{7.20}$$
$$\alpha(\neg(\phi \mathbf{U} \psi)) := \alpha(\neg\phi)\mathbf{R}\alpha(\neg\psi) \tag{7.21}$$
$$\alpha(\phi \mathbf{R} \psi) := \alpha(\phi)\mathbf{R}\alpha(\psi) \tag{7.22}$$
$$\alpha(\neg(\phi \mathbf{R} \psi)) := \alpha(\neg\phi)\mathbf{U}\alpha(\neg\psi) \tag{7.23}$$

■

Remark 7.14. It will be silently assumed, that Remark 7.12 is also applicable to the abstraction of eALTL formulae. ■

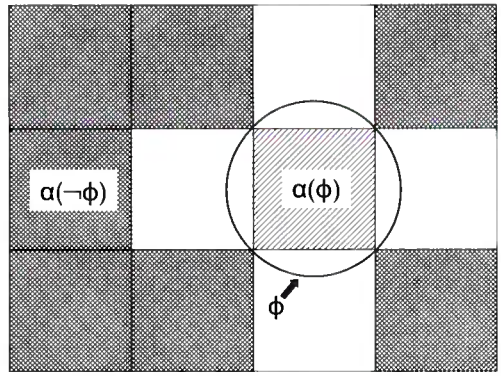

Figure 7.4: Contracting abstraction

Lemma 7.15. The abstraction of action formulae defined in Definition 7.11 is a contracting abstraction. ■

Proof. We prove that $\forall p(q) \in \Lambda : \alpha(p(q)) \models \zeta^\alpha \Rightarrow p(q) \models \zeta$ for an arbitrary property ζ. We prove this by induction on the struture of ζ for an arbitrary single action $p(q)$ for some $p \in A$ and $q \in \mathbb{D}$. We consider four cases for the basic step and two (Cases 5 and 6) for the inductive step. The enumeration of cases is geared to the order of abstraction rules in Definition 7.11.

ad (7.6) $\alpha(p(q)) \models \top \Rightarrow p(q) \models \top$: In this case, the right side trivially holds.

ad (7.7) $\alpha(p(q)) \models \bot \Rightarrow p(q) \models \bot$: In this case, the left side trivially fails.

ad (7.8) $\alpha(p(q)) \models \alpha(\{a(x) \mid expr(x)\}) \Rightarrow p(q) \models \{a(x) \mid expr(x)\}$:

$$\text{Assume: } \alpha(p(q)) \models \alpha(\{a(x) \mid expr(x)\})$$

We say that the abstraction of $p(q)$ holds under the abstraction of ζ if and only if the abstraction of $p(q)$ is an element of the set, which is spanned by the abstraction of ζ:

$$\alpha(p(q)) \in \{a(x^\alpha) \mid \forall x : h_d(x) = x^\alpha \rightarrow expr(x)\}$$

On the level of specifications, the abstraction of $p(q)$ is defined as $\alpha(p(q)) = p(h_d(q))$, since we are using data abstraction:

$$p(h_d(q)) \in \{a(x^\alpha) \mid \forall x : h_d(x) = x^\alpha \rightarrow expr(x)\}$$

At this point, we have to reflect, when $p(h_d(q))$ can actually be an element of the given set. This is the case, if and only if $a = p$ and

$$\forall x : h_d(x) = h_d(q) \rightarrow expr(x)$$

So the expression $expr(x)$ must hold for all possible x for which $h_d(x) = h_d(q)$, thus it also holds for $x = q$. From that, we can conclude:

$$p(q) \in \{a(x) \mid expr(x)\}$$

From this, we can derive following Definition 7.3, that for $a = p$ and $x = q$:

$$p(q) \models \{a(x) \mid expr(x)\}$$

ad (7.9) $\alpha(p(q)) \models \alpha(\neg\{a(x) \mid expr(x)\}) \Rightarrow p(q) \models \neg\{a(x) \mid expr(x)\}$:

$$\text{Assume: } \alpha(p(q)) \models \alpha(\neg\{a(x) \mid expr(x)\})$$

We say that the abstraction of $p(q)$ holds under the abstraction of $\neg\zeta$ if and only if the abstraction of $p(q)$ is an element of the set, which is spanned by the abstraction of $\neg\zeta$:

$$\alpha(p(q)) \in \bigcup_{b(x)} \{\alpha(b(x)) \mid b \neq a\} \cup \{a(x^\alpha) \mid \forall x : h_d(x) = x^\alpha \rightarrow \neg expr(x)\}$$

This means, that $\alpha(p(q))$ must be in one of the two sets. The distinction is made by checking, whether the action name p matches a from the property or not. Since $p(q)$ is either in the first or in the second of the two sets, it holds that that

$$p(q) \in \bigcup_{b(x)} \{b(x) \mid b \neq a\} \cup \{a(x) \mid \neg expr(x)\}$$

This is the complementary set of $\{a(x) \mid expr(x)\}$, so that we can conclude:

$$p(q) \models \neg\{a(x) \mid expr(x)\}$$

We will regard the two resulting cases separately:

a) $p \neq a$: Obviously

$$p(q) \in \bigcup_{b(x)} \{b(x) \mid b \neq a\}$$

and so

$$p(q) \models \neg\{a(x) \mid expr(x)\}.$$

b) $p = a$: In this case, we regard the second set:

$$p(h_d(q)) \in \{a(x^\alpha) \mid \forall x : h_d(x) = x^\alpha \rightarrow \neg expr(x)\}$$

Analogously to Case 2, this is achieved in case $p(h_d(q)) = a(x^\alpha)$, i.e. $a = p$ and

$$\forall x : h_d(x) = h_d(q) \rightarrow \neg expr(x).$$

If the expression $expr(x)$ does not hold for all values of x, it will surely also not hold for $x = q$ so that we can conclude:

$$p(q) \in \{a(x) \mid \neg expr(x)\},$$

so that

$$p(q) \models \neg\{a(x) \mid expr(x)\}.$$

ad (7.11) $\alpha(p(q)) \models \alpha(\zeta_1 \vee \zeta_2) \Rightarrow p(q) \models \zeta_1 \vee \zeta_2$:

This is the second inductive step. Assume, that $\alpha(p(q)) \models \alpha(\zeta_1 \vee \zeta_2)$. Then, according to the definition, $\alpha(p(q)) \models \alpha(\zeta_1) \vee \alpha(\zeta_2)$ holds, from which we can derive, that $\alpha(p(q)) \models \alpha(\zeta_1)$ or $\alpha(p(q)) \models \alpha(\zeta_2)$. By the induction hypothesis, we can conclude that then $p(q) \models \zeta_1$ or $p(q) \models \zeta_2$ and thus $p(q) \models \zeta_1 \vee \zeta_2$.

ad (7.12) $\alpha(p(q)) \models \alpha(\zeta_1 \wedge \zeta_2) \Rightarrow p(q) \models \zeta_1 \wedge \zeta_2$:

This is the inductive step. Assume, that $\alpha(p(q)) \models \alpha(\zeta_1 \wedge \zeta_2)$. Then, according to the definition, $\alpha(p(q)) \models \alpha(\zeta_1) \wedge \alpha(\zeta_2)$ holds, from which we can derive, that $\alpha(p(q)) \models \alpha(\zeta_1)$ and $\alpha(p(q)) \models \alpha(\zeta_2)$. By the induction hypothesis, we can conclude that then $p(q) \models \zeta_1$ and $p(q) \models \zeta_2$ and thus $p(q) \models \zeta_1 \wedge \zeta_2$.

□

Lemma 7.16. The abstraction of eALTL formulae defined in Definition 7.13 is contracting. ■

Proof. We prove that $\forall \pi \in \Lambda^* : \alpha(\pi) \models \phi^\alpha \Rightarrow \pi \models \phi$ for an arbitrary property ϕ. We do this by induction on the structure of ϕ. We consider traces of one or more steps. Thus, we consider inductive steps as an extension of the previous proof. The enumeration of the cases is geared to the order of abstraction rules in Definition 7.13.

Action formulae. ϕ is an action formula. This case has been proven in Lemma 7.15.

ad (7.14) $\alpha(\pi) \models \alpha(\Box\phi) \Rightarrow \pi \models \Box\phi$:

Assume that $\alpha(\pi) \models \alpha(\Box\phi)$. Then, by definition, $\alpha(\pi) \models \Box\alpha(\phi)$. This means, that $\forall i \in \mathbb{N} \setminus \{0\} : \alpha(\pi)^i \models \alpha(\phi)$. By induction hypothesis, we can then claim, that also $\forall i \in \mathbb{N} \setminus \{0\} : \pi^i \models \phi$ and thus $\pi \models \Box\phi$.

ad (7.15) $\alpha(\pi) \models \alpha(\neg\Box\phi) \Rightarrow \pi \models \neg\Box\phi$:

As it has been stated earlier in this chapter, $\Box\phi \equiv \neg\Diamond\neg\phi$. From this, we can immediately derive, that $\neg\Box\phi \equiv \neg\neg\Diamond\neg\phi \equiv \Diamond\neg\phi$. This means, that $\alpha(\neg\Box\phi) \equiv \alpha(\Diamond\neg\phi)$. As defined in Definition 7.13, this is equivalent to an abstraction $\Diamond\alpha(\neg\phi)$.

Now assume, that $\alpha(\pi) \models \alpha(\neg\Box\phi) \Leftrightarrow \alpha(\pi) \models \Diamond\alpha(\neg\phi)$. As we will prove in Case 4, then $\pi \models \Diamond\neg\phi$. Due to the derivation earlier in this case, this is equivalent to $\pi \models \neg\Box\phi$.

ad (7.16) $\alpha(\pi) \models \alpha(\Diamond\phi) \Rightarrow \pi \models \Diamond\phi$:

Assume that $\alpha(\pi) \models \alpha(\Diamond\phi)$. This is equivalent to $\alpha(\pi) \models \alpha(\top \mathbf{U} \phi)$. This can be abstracted as follows: $\alpha(\pi) \models \alpha(\top)\mathbf{U}\alpha(\phi)$ and further to $\alpha(\pi) \models \top\mathbf{U}\alpha(\phi)$ and finally $\alpha(\pi) \models \Diamond\alpha(\phi)$.

As will be proven later, it holds that $\alpha(\pi) \models \alpha(\top)\mathbf{U}\alpha(\phi) \Rightarrow \pi \models \top\mathbf{U}\phi$. Remembering that $\top\mathbf{U}\phi$ is equivalent to $\Diamond\phi$, we can conclude, that $\pi \models \Diamond\phi$.

ad (7.17) $\alpha(\pi) \models \alpha(\neg\Diamond\phi) \Rightarrow \pi \models \neg\Diamond\phi$:

As it has been stated earlier in this chapter, $\neg\Diamond\neg\phi \equiv \Box\phi$. From this, we can immediately derive, that $\neg\Diamond\phi \equiv \neg\Diamond\neg\neg\phi \equiv \Box\neg\phi$. This means, that $\alpha(\neg\Diamond\phi) \equiv$

$\alpha(\Box\neg\phi)$. As defined in Definition 7.13, this is equivalent to an abstraction $\Box\alpha(\neg\phi)$.

Now assume, that $\alpha(\pi) \models \alpha(\neg\Diamond\phi) \Leftrightarrow \alpha(\pi) \models \Box\alpha(\neg\phi)$. As we have proven in Case 2, then $\pi \models \Box\neg\phi$. Due to the derivation earlier in this case, this is equivalent to $\pi \models \neg\Diamond\phi$.

ad (7.18) $\alpha(\pi) \models \alpha(\phi \wedge \psi) \Rightarrow \pi \models \phi \wedge \psi$:

Assume, that $\alpha(\pi) \models \alpha(\phi \wedge \psi)$. Then, according to the definition, $\alpha(\pi) \models \alpha(\phi) \wedge \alpha(\psi)$ holds, from which we can derive, that $\alpha(\pi) \models \alpha(\phi)$ and $\alpha(\pi) \models \alpha(\psi)$. By the induction hypothesis, we can conclude that then $\pi \models \phi$ and $\pi \models \psi$ and thus $\pi \models \phi \wedge \psi$.

ad (7.19) $\alpha(\pi) \models \alpha(\phi \vee \psi) \Rightarrow \pi \models \phi \vee \psi$:

Assume, that $\alpha(\pi) \models \alpha(\phi \vee \psi)$. Then, according to the definition, $\alpha(\pi) \models \alpha(\phi) \vee \alpha(\psi)$ holds, from which we can derive, that $\alpha(\pi) \models \alpha(\phi)$ or $\alpha(\pi) \models \alpha(\psi)$. By the induction hypothesis, we can conclude that then $\pi \models \phi$ or $\pi \models \psi$ and thus $\pi \models \phi \vee \psi$.

ad (7.20) $\alpha(\pi) \models \alpha(\phi \mathbf{U} \psi) \Rightarrow \pi \models \phi \mathbf{U} \psi$:

Assume, that $\alpha(\pi) \models \alpha(\phi \mathbf{U} \psi)$. Then, following Definition 7.13, $\alpha(\pi) \models \alpha(\phi)\mathbf{U}\alpha(\psi)$. This means, that there exists a $k \in \mathbb{N} \setminus \{0\}$ such that for all $0 < i < k : \alpha(\pi)^i \models \alpha(\phi)$ and $\alpha(\pi)^k \models \alpha(\psi)$.

The abstraction of traces as defined in Definition 7.8 does not affect the indices of steps in a trace. This means, that when we follow the induction hypothesis, we can assume, that $\alpha(\pi)[i] = \alpha(\pi[i])$ for some $i \in \mathbb{N} \setminus \{0\}$ and thus $\alpha(\pi)^i = \alpha(\pi^i)$. Hence, for all $0 < i < k : \pi^i \models \phi$ and $\pi^k \models \psi$. From that, we can immediately derive that $\pi \models \phi \mathbf{U} \psi$.

ad (7.21) $\alpha(\pi) \models \alpha(\neg(\phi \mathbf{U} \psi)) \Rightarrow \pi \models \neg(\phi \mathbf{U} \psi)$:

Assume, that $\alpha(\pi) \models \alpha(\neg(\phi \mathbf{U} \psi))$. Then it logically holds that

$$\alpha(\pi) \models \alpha(\neg\neg(\neg\phi \mathbf{R} \neg\psi)) \Leftrightarrow \alpha(\pi) \models \alpha(\neg\phi \mathbf{R} \neg\psi).$$

By induction hypothesis, we derive that then also

$$\pi \models \neg\phi \mathbf{R} \neg\psi \Leftrightarrow \pi \models \neg\neg(\neg\phi \mathbf{R} \neg\psi) \Leftrightarrow \pi \models \neg(\phi \mathbf{U} \psi).$$

ad (7.22) $\alpha(\pi) \models \alpha(\phi \mathbf{R} \psi) \Rightarrow \pi \models \phi \mathbf{R} \psi$:

Assume, that $\alpha(\pi) \models \alpha(\phi \mathbf{R} \psi)$. Then, following Definition 7.13, $\alpha(\pi) \models \alpha(\phi)\mathbf{R}\alpha(\psi)$. This means, that either $\forall i \in \mathbb{N} \setminus \{0\} : \alpha(\psi)$ or there exists a $k \in \mathbb{N} \setminus \{0\}$ such that for all $0 < i < k : \alpha(\pi)^i \models \alpha(\psi)$ and $\alpha(\pi)^k \models \alpha(\phi)$.

The abstraction of traces as defined in Definition 7.8 does not affect the indices of steps in a trace. This means, that when we follow the induction hypothesis, we can assume, that $\alpha(\pi)[i] = \alpha(\pi[i])$ for some $i \in \mathbb{N} \setminus \{0\}$ and thus $\alpha(\pi)^i = \alpha(\pi^i)$. Hence, $\forall i \in \mathbb{N} \setminus \{0\} : \pi^i \models \psi$ or there exists a $k \in \mathbb{N} \setminus \{0\}$ such that for all $0 < i < k : \pi^i \models \psi$ and $\pi^k \models \phi$. From that, we can immediately derive that $\pi \models \phi \mathbf{R} \psi$.

ad (7.23) $\alpha(\pi) \models \alpha(\neg(\phi \mathbf{R} \psi)) \Rightarrow \pi \models \neg(\phi \mathbf{R} \psi)$:

Assume, that $\alpha(\pi) \models \alpha(\neg(\phi \mathbf{R} \psi))$. Then it logically holds that

$$\alpha(\pi) \models \alpha(\neg\neg(\neg\phi \mathbf{U} \neg\psi)) \Leftrightarrow \alpha(\pi) \models \alpha(\neg\phi \mathbf{U} \neg\psi).$$

By induction hypothesis, we derive that then also

$$\pi \models \neg\phi \mathbf{U} \neg\psi \Leftrightarrow \pi \models \neg\neg(\neg\phi \mathbf{U} \neg\psi) \Leftrightarrow \pi \models \neg(\phi \mathbf{R} \psi).$$

□

In order to have precise abstractions, we need a restriction on the homomorphism $\alpha = \langle h_s, h_a \rangle$. We define that α is *consistent* with ϕ, if and only if for all action formulae ζ occuring in ϕ, $\{h_a(act) | act \models \zeta\} \cap [\![\neg\alpha(\zeta)]\!] = \emptyset$, i.e. the hatched square and the ellipse in Figure 7.4 coincide.

Lemma 7.17. If α is consistent with ϕ, then $\alpha(\phi)$ is precise. ■

Proof. We prove by induction on the structure of the eALTL formula ϕ that

$$\underbrace{\left(\forall \zeta \in \phi : \{h_a(p(q)) \mid p(q) \models \zeta\} \cap [\![\neg\alpha(\zeta)]\!] = \emptyset\right)}_{\text{consistency}} \Rightarrow \underbrace{\left(\forall \pi \in \Lambda^\star : \pi^\alpha \models \alpha(\phi) \Leftrightarrow \pi \models \phi\right)}_{\text{precision}}$$

Action Formulae. We begin with the case that ϕ is an action formula ζ and that $\pi = p(q)$. We assume, that α is consistent with ϕ, i.e. with ζ. We have to show that $\alpha(p(q)) \models \zeta^\alpha \Leftrightarrow p(q) \models \zeta$. We distinguish two cases:

a) $\alpha(p(q)) \models \zeta^\alpha \Rightarrow p(q) \models \zeta$: This case follows directly from Lemma 7.15.

b) $\alpha(p(q)) \models \zeta^\alpha \Leftarrow p(q) \models \zeta$: Let us assume, that $p(q) \models \zeta \wedge \alpha(p(q)) \not\models \zeta^\alpha$. In this case, due to the first conjunct $h_a(p(q)) \in \{h_a(p(q)) \mid p(q) \models \zeta\}$ and due to the second one $h_a(p(q)) \in [\![\neg\alpha(\zeta)]\!]$. The intersection of both sets is thus not equal, what contradicts our assumption and proves the hypothesis correct.

ad (7.14) $\alpha(\pi) \models \alpha(\Box\phi) \Leftarrow \pi \models \Box\phi$:

Assume that $\pi \models \Box\phi$. This means, that $\forall i \in \mathbb{N} \setminus \{0\} : \pi^i \models \phi$. By induction hypothesis, we can then claim, that also $\forall i \in \mathbb{N} \setminus \{0\} : \alpha(\pi)^i \models \alpha(\phi)$ and thus $\alpha(\pi) \models \alpha(\Box\phi)$.

ad (7.15) $\alpha(\pi) \models \alpha(\neg\Box\phi) \Leftarrow \pi \models \neg\Box\phi$:

As it has been stated earlier in this chapter, $\Box\phi \equiv \neg\Diamond\neg\phi$. From this, we can immediately derive, that $\neg\Box\phi \equiv \neg\neg\Diamond\neg\phi \equiv \Diamond\neg\phi$. This means, that $\alpha(\neg\Box\phi) \equiv \alpha(\Diamond\neg\phi)$. As defined in Definition 7.13, this is equivalent to $\Diamond\alpha(\neg\phi)$.

Now assume, that $\pi \models \neg\Box\phi$. According to the above, this means that $\pi \models \Diamond\neg\phi$. As we will prove in Case 4, then $\alpha(\pi) \models \alpha(\Diamond\neg\phi)$. Due to the derivation earlier in this case, this is equivalent to $\alpha(\pi) \models \alpha(\neg\Box\phi)$.

ad (7.16) $\alpha(\pi) \models \alpha(\Diamond\phi) \Leftarrow \pi \models \Diamond\phi$:

Assume that $\pi \models \Diamond\phi$. This means that there exists an $i \in \mathbb{N} \setminus \{0\}$, such that $\pi^i \models \phi$. The abstraction of traces according to Definition 7.8 does not affect the indices of steps in a trace. By induction hypothesis, we receive thus $\exists i \in \mathbb{N} \setminus \{0\} : \alpha(\pi)^i \models \alpha(\phi)$. This, however, leads directly to $\alpha(\pi) \models \Diamond\alpha(\phi)$ and according to Definition 7.13 to $\alpha(\pi) \models \alpha(\Diamond\phi)$.

ad (7.17) $\alpha(\pi) \models \alpha(\neg\Diamond\phi) \Leftarrow \pi \models \neg\Diamond\phi$:

As it has been stated earlier in this chapter, $\neg\Diamond\neg\phi \equiv \Box\phi$. From this, we can immediately derive, that $\neg\Diamond\phi \equiv \neg\Diamond\neg\neg\phi \equiv \Box\neg\phi$. This means, that $\alpha(\neg\Diamond\phi) \equiv \alpha(\Box\neg\phi)$. As defined in Definition 7.13, this is equivalent to $\Box\alpha(\neg\phi)$.

Now assume, that $\pi \models \neg\Diamond\phi$. As we have proven in Case 2, then $\alpha(\pi) \models \Box\alpha(\neg\phi)$. Due to the derivation earlier in this case, this is equivalent to $\alpha(\pi) \models \alpha(\neg\Diamond\phi)$.

ad (7.18) $\alpha(\pi) \models \alpha(\phi \wedge \psi) \Leftarrow \pi \models \phi \wedge \psi$:

Assume that $\pi \models \phi \wedge \psi$. This means, that $\pi \models \phi$ as well as $\pi \models \psi$ hold. Following the induction hypothesis, we can claim that $\alpha(\pi) \models \alpha(\phi)$ and $\alpha(\pi) \models \alpha(\psi)$ and thus $\alpha(\pi) \models \alpha(\phi \wedge \psi)$ holds.

ad (7.19) $\alpha(\pi) \models \alpha(\phi \vee \psi) \Leftarrow \pi \models \phi \vee \psi$:

Assume that $\pi \models \phi \vee \psi$. This means, that $\pi \models \phi$ or $\pi \models \psi$ holds. Following the induction hypothesis, we can claim that $\alpha(\pi) \models \alpha(\phi)$ or $\alpha(\pi) \models \alpha(\psi)$ and thus $\alpha(\pi) \models \alpha(\phi \vee \psi)$ holds.

ad (7.20) $\alpha(\pi) \models \alpha(\phi \mathbf{U} \psi) \Leftarrow \pi \models \phi \mathbf{U} \psi$:

Assume that $\pi \models \phi \mathbf{U} \psi$. This means, that $\exists k \in \mathbb{N} \setminus \{0\}$ such that $\forall i, 0 < i < k : \pi^i \models \phi$ and $\pi^k \models \psi$ hold. Following the induction hypothesis and since $\forall i \in \mathbb{N} \setminus \{0\} : \alpha(\pi)[i] = \alpha(\pi[i])$, we can claim that $\exists k \in \mathbb{N} \setminus \{0\}$ such that $\forall i, 0 < i < k : \alpha(\pi)^i \models \alpha(\phi)$ and $\alpha(\pi)^k \models \alpha(\psi)$ and thus $\alpha(\pi) \models \alpha(\phi \mathbf{U} \psi)$ holds.

ad (7.21) $\alpha(\pi) \models \alpha(\neg(\phi \mathbf{U} \psi)) \Leftarrow \pi \models \neg(\phi \mathbf{U} \psi)$:

Assume that $\pi \models \neg(\phi \mathbf{U} \psi)$. This means, that $\pi \models \neg\neg(\neg\phi \mathbf{R} \neg\psi)$ and thus $\pi \models \neg\phi \mathbf{R} \neg\psi$ hold. Following the induction hypothesis we can claim that then also $\alpha(\pi) \models \alpha(\neg\phi \mathbf{R} \neg\psi)$ and resulting from that $\alpha(\pi) \models \alpha(\neg\neg(\neg\phi \mathbf{R} \neg\psi))$ and $\alpha(\pi) \models \alpha(\neg(\phi \mathbf{U} \psi))$ holds.

ad (7.22) $\alpha(\pi) \models \alpha(\phi \mathbf{R} \psi) \Leftarrow \pi \models \phi \mathbf{R} \psi$:

Assume that $\pi \models \phi \mathbf{R} \psi$. This means, that for all $k \in \mathbb{N} \setminus \{0\}$ holds that if for all $i \in \mathbb{N} \setminus \{0\}$ with $i < k$ we have if $\pi^i \not\models \phi$, then $\pi^i \models \psi$.

The abstraction of traces according to Definition 7.8 does not affect the indices of steps in a trace. This means, that when we follow the induction hypothesis, we can assume, that $\alpha(\pi[i]) = \alpha(\pi)[i]$ for some $i \in \mathbb{N} \setminus \{0\}$ and thus $\alpha(\pi^i) = \alpha(\pi)^i$. Hence, $\forall k \in \mathbb{N} \setminus \{0\}$ we have that if $\forall i \in \mathbb{N} \setminus \{0\}.i < k : \alpha(\pi)^i \models \alpha(\phi) \vee (\alpha(\pi)^i \not\models \alpha(\phi) \wedge \alpha(\pi)^i \models \alpha(\psi))$. From that, we can immediately derive that $\alpha(\pi) \models \alpha(\phi \mathbf{R} \psi)$.

ad (7.23) $\alpha(\pi) \models \alpha(\neg(\phi \mathbf{R} \psi)) \Leftarrow \pi \models \neg(\phi \mathbf{R} \psi)$:

Assume that $\pi \models \neg(\phi \mathbf{R} \psi)$. This means, that $\pi \models \neg\neg(\neg\phi \mathbf{U} \neg\psi)$ and thus $\pi \models \neg\phi \mathbf{U} \neg\psi$ hold. Following the induction hypothesis we can claim that then also $\alpha(\pi) \models \alpha(\neg\phi \mathbf{U} \neg\psi)$ and resulting from that $\alpha(\pi) \models \alpha(\neg\neg(\neg\phi \mathbf{U} \neg\psi))$ and $\alpha(\pi) \models \alpha(\neg(\phi \mathbf{R} \psi))$ holds.

□

7.4 Classes of Counterexamples

We can now explain model checking by abstraction for eALTL formulae. Let a specification $\mathfrak{S}$ (with an underlying LTS $\mathfrak{M}$) and an eALTL property ϕ be given. Let us investigate whether a contracting abstraction α suffices for our needs. We compute $\alpha(\phi)$ and $\mathfrak{S}^\alpha$, generate its underlying LTS $\mathfrak{M}^\alpha$ and use a model checking algorithm to check $\mathfrak{M}^\alpha \models \phi^\alpha$. If this holds, we can derive by our previous results, that also $\mathfrak{M} \models \phi$, without ever generating $\mathfrak{M}$. If it does not hold, we obtain a counterexample. Here we provide a classification of abstract counterexamples, and demonstrate their relationship with contracting and precise abstractions of eALTL formulae.

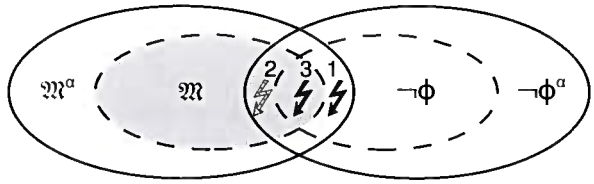

Figure 7.5: Classification of counterexamples

Given a concrete system $\mathfrak{M}$, its abstraction $\mathfrak{M}^\alpha$, a property ϕ and its abstraction ϕ^α, we differentiate between three classes of abstract counterexamples (see Figure 7.5). Given a counterexample χ^α, we refer to a concrete trace $\chi \in [\![\mathfrak{M}]\!]_{\text{traces}}$ such that $\chi^\alpha = \alpha(\chi)$ as a *concrete counterpart* of χ^α. The first class (see counterexample 1 in Figure 7.5) consists of the counterexamples having *no* concrete counterparts in the concrete system. These counterexamples are referred to as *false negatives*.

The second class (see counterexample 2 in Figure 7.5) consists of counterexamples having (at least) one concrete counterpart *satisfying* the original property. We further refer to this class as *spurious counterexamples*.

The third class (see counterexample 3 in Figure 7.5) consists of the counterexamples having at least one counterpart in the concrete system; moreover all concrete counterparts violate the concrete property. Counterexamples from this class are referred to as *ideal counterexamples*.

Definition 7.18. Let χ^α be a counterexample obtained by verifying an abstraction ϕ^α of a property ϕ on the abstraction $\mathfrak{M}^\alpha$ of a system $\mathfrak{M}$ w.r.t. the homomorphism h. We distinguish the following three cases:

1. We call χ^α a *false negative*, if there is no $\chi \in [\![\mathfrak{M}]\!]_{\text{traces}}$ such that $\chi^\alpha = \alpha(\chi)$.
2. We call χ^α a *spurious* counterexample if there exists $\chi \in [\![\mathfrak{M}]\!]_{\text{traces}}$ such that $\chi^\alpha = \alpha(\chi)$ and $\chi \models \phi$.
3. Otherwise, we call χ^α an *ideal* counterexample.

■

Contracting abstractions may lead to spurious counterexamples, as illustrated below.

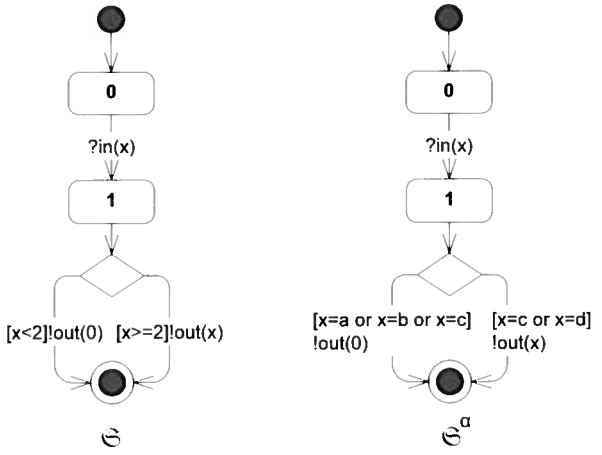

Figure 7.6: Concrete and abstracted specifications from Example 7.19

Example 7.19. Let $\mathfrak{S}$ in Figure 7.6 be the specification of a concrete system. We abstract $\mathbb{Z}$ into $\mathbb{Z}^\alpha = \{a, b, c, d\}$, where a stands for the integers from $]-\infty, -3[$; b stands for the integers from $[-3, 0]$; c stands for the integers from $]0, 3]$; and d stands for the integers from $]3, +\infty[$. By applying this abstraction to $\mathfrak{S}$ we obtain $\mathfrak{S}^\alpha$ (see Figure 7.6).

Consider the property $\phi = \Diamond(\{\text{out}(x) \mid (x \geqslant 2)\})$. We compute the contracting abstraction of ϕ as follows:

$$\begin{aligned} \phi &= \Diamond(\{\text{out}(x) \mid (x \geqslant 2)\}) \\ \phi^\alpha &= \Diamond(\{\text{out}(x^\alpha) \mid \forall x : h_d(x) = x^\alpha \rightarrow (x \geqslant 2)\}) \\ &= \Diamond(\text{out}(d)) \end{aligned}$$

Verifying ϕ^α on $\mathfrak{S}^\alpha$ we may obtain the trace $\text{in}(c).\text{out}(c)$ as a counterexample, because it is a trace in $\mathfrak{S}^\alpha$, but it does not satisfy ϕ. However, the concrete traces

$in(2).out(2)$ and $in(3).out(3)$ corresponding to the abstract counterexample satisfy $\Diamond(out(x) \wedge (x \geqslant 2))$. Hence, $\neg\phi^\alpha$ is not precise enough.

Such spurious counterexamples are problematic for tracking real bugs. Therefore, we will use *precise* abstractions, in order to avoid spurious counterexamples. A contracting abstraction can be made precise, by fitting the abstraction to the predicates in the specification and the formula:

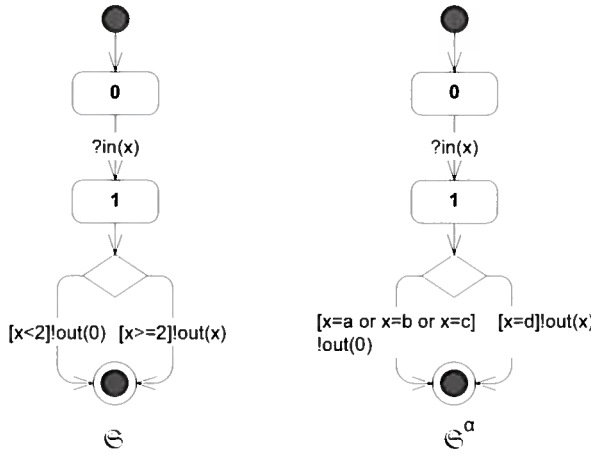

Figure 7.7: Concrete and abstracted specifications from Example 7.20

Example 7.20. Let $\mathfrak{S}$ in Figure 7.7 be the specification of a concrete system. We abstract $\mathbb{Z}$ into $\mathbb{Z}^\alpha = \{a, b, c, d\}$ where the interpretation of a and b remains the same as in Example 7.19 while c represents the integers from the interval $]0, 2[$ and d represents those from $[2, +\infty[$. By applying this abstraction to $\mathfrak{S}$ we obtain $\mathfrak{S}^\alpha$ (see Figure 7.7).

Consider again the property $\phi = \Diamond(\{out(x) \mid (x \geqslant 2)\})$ and its abstraction $\phi^\alpha = \Diamond(out(d))$. Verifying ϕ^α on $\mathfrak{S}^\alpha$ we may obtain the following counterexamples: $in(a).out(b)$, $in(b).out(b)$, and $in(c).out(b)$. In this example it is straightforward to see that any concretization of these traces is a counterexample for ϕ. So in this case, the abstraction is precise.

7.5 Constructing a Violation Pattern

Counterexamples that are false negatives still have a value for detecting bugs in specifications. By relaxing them, i.e. making them even more abstract, false negatives cover a larger part of the system, which can contain bugs. In this manner, they can serve as a starting point for bug hunting.

In this section, we provide an overview of our framework for bug hunting with false negatives. This process comprises the following steps:

1. Specify a requirement as a formula ϕ of eALTL.
2. Choose and apply a data abstraction, which is consistent with ϕ, to the specification of the concrete system and to the concrete property.
3. Abstract counterexamples for the property are (automatically) determined using model checking.
4. If a false negative is found, generalize it further by *relaxing* actions that are not directly relevant for our search. This results in a violation pattern. The relaxing process itself is automatic, only the counterexample and the set of directly relevant actions have to be given as input to the algorithm (see Algorithm 7.1).
5. The concrete counterexamples are automatically computed by finding the intersection of the original system and the violation pattern.

Since the first three steps of the framework can be handled by existing data abstraction and model checking techniques, our contribution concerns the steps 4 and 5 of the framework.

7.5.1 Constructing a Violation Pattern

A counterexample that we obtain in case the property is violated on our abstract model is an infinite trace of the form $\pi_p\pi_s^\omega$, where π_p is a finite prefix and π_s^ω is a cyclic suffix with a finite *cycle base* π_s.

Although the counterexample χ^α may have no counterpart in the concrete system, it can contain a clue about a counterexample present in the concrete system. Therefore we transform a counterexample χ^α into a *violation pattern* $\mathfrak{V}$, considering only *infinite* counterexamples.

A violation pattern is an LTS that accepts only traces hitting a distinguished *cyclic* state infinitely often. The violation pattern accepts only traces which are *similar* to the counterexample and *violate* the abstract property. The actions mentioned in the property are essential for the property violation. Therefore, we keep at least this information in the violation pattern. In order to support this kind of properties, we also keep this information in the violation pattern. For actions influenced by abstraction, the order and the number of actions in a similar trace may differ from those in the counterexample. We will first illustrate the idea of similarity on a simple example and then generalize it.

Example 7.21. Let us come back to the example from the introduction. Assume that we model-check the property $\Box(a \rightarrow \Diamond b)$ and obtain the abstract counterexample $a.set(k^+).tick^3.b.(a.set(k^+).tick^2.d)^\omega$ (see Figure 7.8). The k^+ is in this case an abstraction of a timer: The original value of the timer is preserved up to k; any value above k is abstracted to the constant value k^+. To guarantee that the property is violated by any trace accepted by the pattern, we keep *at least* the actions a and b, because they are mentioned in the property. Since we are searching for similar

traces with an infinite *cyclic* suffix π_s, we may also decide to keep information about some actions of this cycle. Here we also preserve the action step d in the cycle (see Figure 7.9). The actions tick and set(k^+) are not mentioned in the property, and are definitely influenced by the timer abstraction. Therefore, we *relax* these actions, meaning, we allow these actions to occur an arbitrary number of times in an arbitrary order; however, at least one of these actions has to appear once (see states 1 and 5 of the violation pattern in Figure 7.9 and then states 2 and 7 for the self-loops). In order to prevent self-loops in the cyclic state (state 4), we insert a τ-step between respectively states 3 or 7 and the cyclic state. In states 3 and 7, self-loops are in principle allowed, but this is not applicable to this example. As we will see later, this step is necessary for the validity of our theory. At this point, it should also be remarked, that $\tau \notin \Lambda_{keep}$.

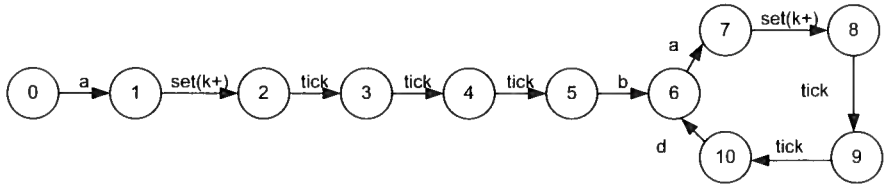

Figure 7.8: A concrete counterexample

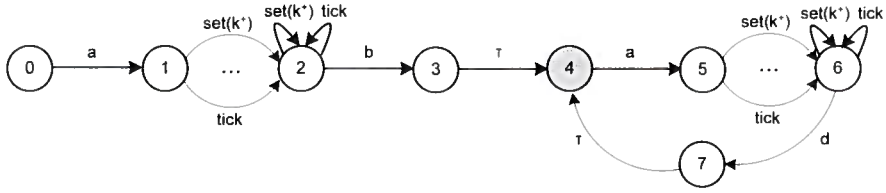

Figure 7.9: The violation pattern for the counterexample

We refer to the set of action labels that we do not want to relax by Λ_{keep}. This set includes *at least* all the labels mentioned in the abstract (and also the concrete) property. In the violation pattern, we distinguish a *cyclic* state that corresponds to the first state in the cyclic suffix. The last action in the cycle base of an infinite counterexample leads to this cyclic state.

In principle, we would like to relax more actions influenced by data abstraction. These actions can be found by applying static analysis techniques. The more actions we keep, the more concrete the counterexample is and the faster we can check whether there is a concrete trace matching the pattern. By keeping too many actions, however, we might end up with a violation pattern that specifies traces having no counterparts in the concrete system.

Definition 7.22 (Non-relaxed Actions). Given a set A of actions appearing in a property ϕ^α. We define that some set Λ_{keep} of non-relaxed actions in a violation pattern is *consistent* with ϕ^α if and only if $\Lambda_{keep} \supseteq A$. ■

Λ_{keep} can optionally contain additional actions, like the last action of a cyclic suffix, or actions not influenced by the data abstraction, to make the violation pattern more specific.

Definition 7.23 (Violation Pattern). Given an abstract counterexample $\chi^\alpha = \pi_p \pi_s^\omega$ and a set Λ_{keep} of non-relaxed actions, a *violation pattern* is an extended LTS $\mathfrak{V} = (\Sigma, \Lambda, \Delta, \sigma_{init}, \sigma_{cyclic})$ constructed by Algorithm 7.1, where σ_{cyclic} is the cyclic state.

The set of traces visiting the cyclic state infinitely often, is further referred to as the set $[\![\mathfrak{V}]\!]_{traces}$ of *accepted traces*. ■

Algorithm 7.1 Build Violation Pattern

Require: $\chi^\alpha = \pi_p \pi_s^\omega, \Lambda_{keep}$ // *trace, actions to keep*
Ensure: $\mathfrak{V} = (\Sigma, \Lambda, \Delta, \sigma_{init}, \sigma_{cyclic})$ // *violation pattern*

```
1  σ_init := 0; Σ := {σ_init};                          // initialization
2  σ := σ_init;                                         // current state σ of 𝔙
3  for all i = 1..|π_p π_s| do                          // for all steps of π_p π_s
4    if χ^α[i] ∉ Λ_keep then
5      if σ = σ_init ∨ χ^α[i − 1] ∈ Λ_keep then       // prev. action in Λ_keep: add loop state
6        σ̂ := σ + 1;
7        Σ := Σ ∪ {σ̂};
8      fi
9      Δ := Δ ∪ {(σ, χ^α[i], σ̂), (σ̂, χ^α[i], σ̂)};     // add a relaxed step
10   else                                               // if step to be kept
11     σ̂ := σ + 1;                                     // next state is arbitrary
12     Σ := Σ ∪ {σ̂};                                   // add the new state
13     Δ := Δ ∪ {(σ, χ^α[i], σ̂)};                      // add the step to the next state
14   fi
15   σ := σ̂;                                          // proceed with the next state of 𝔙
16   if i = |π_p| + 1 then                             // assignment of σ_cyclic
17     σ_cyclic := σ̂ + 1;
18     Δ := Δ ∪ {(σ̂, τ, σ_cyclic)};                    // add the internal step to the cyclic state
19     σ := σ_cyclic;
20   fi
21 od
22 Δ := Δ ∪ {(σ, τ, σ_cyclic)};                        // add the internal step to the cyclic state
```

Given a counterexample $\chi^\alpha = \pi_p \pi_s^\omega$ and a set Λ_{keep} of actions to keep, Algorithm 7.1 constructs the violation pattern $\mathfrak{V}$. The algorithm starts with creating the initial state $\sigma_{init} := 0$ of $\mathfrak{V}$ and goes through $\pi_p \pi_s$. When the algorithm encounters an action to relax, it adds a single transition labeled with this action from the previous to the current state, followed by an equally-labeled self-loop transition in the current state

of $\mathfrak{V}$. When it encounters an action to keep, it adds a transition from the current state to the (new) next state labeled with this action. When the algorithm has reached the end of the cycle base, a τ-step leads back to the cyclic state. The first state of π_s is assigned to σ_{cyclic}.

Next, we show that all traces obtained from the traces of $[\![\mathfrak{V}]\!]_{traces}$ violate the property ϕ^{α}. Therefore, we first introduce a function that projects traces on Λ_{keep}, and then prove the according relations between traces and properties. By trace projection, a trace π is transformed into a second trace $\lfloor\pi\rfloor_{\Lambda_{keep}}$, which preserves all non-relaxed actions and for each sequence of relaxed actions the first action of this sequence. A trace $\pi = a.b.b.a.c.b.a$ with $b \notin \Lambda_{keep}$ and $a, c \in \Lambda_{keep}$ is, for instance, transformed to a trace $\lfloor\pi\rfloor_{\Lambda_{keep}} = a.\tau.a.c.\tau.a$. It is obvious that the actions from Λ_{keep} must be kept. The reason to keep some of the relaxed actions can be shown with the property $\phi = \Box(a \vee c)$, which prohibits the occurrence of b. The action b is officially a relaxed action, since it does not occur in ϕ. However, completely removing it from $\lfloor\pi\rfloor_{\Lambda_{keep}}$ would violate the soundness of our approach, since then $\pi \not\models \phi$ while $\lfloor\pi\rfloor_{\Lambda_{keep}} \models \phi$.

Definition 7.24 (Projection of Traces on Λ_{keep}). Let $\mathfrak{M} = (\Sigma, \Lambda, \Delta, \sigma_{init})$ be an LTS and $\pi \in [\![\mathfrak{M}]\!]_{traces}$ be an arbitrary trace. $\lfloor\pi\rfloor_{\Lambda_{keep}}$ is this trace projected on Λ_{keep} by a projection function $p_\pi : \mathbb{N}\backslash\{0\} \to \mathbb{N}\backslash\{0\}$, such that

$$\forall i \in \mathbb{N} : p_\pi(i+1) = \begin{cases} 1 & \iff i = 0 \\ p_\pi(i) & \iff i > 0 \wedge (\pi[i+1] \notin \Lambda_{keep} \wedge \pi[i] \notin \Lambda_{keep}) \\ p_\pi(i)+1 & \iff i > 0 \wedge (\pi[i+1] \in \Lambda_{keep} \vee \pi[i] \in \Lambda_{keep}) \end{cases}$$

We furthermore define that $\forall p_\pi(i) \in \mathbb{N}\backslash\{0\}$

$$\lfloor\pi\rfloor_{\Lambda_{keep}}[p_\pi(i)] = \begin{cases} \pi[i] & \iff \pi[i] \in \Lambda_{keep} \\ \tau & \iff \pi[i] \notin \Lambda_{keep} \end{cases}$$

with $\tau \notin \Lambda$. ■

The function p_π is surjective, i.e. every element of the result set of p_π has at least one preimage. Since this would not be given for traces π, whose steps from some point on are only outside of Λ_{keep}, we assume, that in this case infinitely many τ-steps are added to $\lfloor\pi\rfloor_{\Lambda_{keep}}$ in order to preserve the surjectivity of p_π. In some cases, where it is clear which trace is projected by p_π, we will leave out the subscript in order to improve readability.

In the following, we have to define two invariances of properties, regarding a trace π and its projected counterpart $\lfloor\pi\rfloor_{\Lambda_{keep}}$. The first invariance states, that a property is invariant if either both traces π and $\lfloor\pi\rfloor_{\Lambda_{keep}}$ satisfy it, or both traces do not. The second invariance definition follows in Definition 7.28.

Definition 7.25 (Invariance of Properties (1)). A property ϕ is invariant under the projection p_π of π to Λ_{keep}, if and only if $\pi \models \phi \Leftrightarrow \lfloor\pi\rfloor_{\Lambda_{keep}} \models \phi$. ■

Lemma 7.26. Any eALTL property ϕ is invariant under projection p_π of trace π to Λ_{keep}, provided that Λ_{keep} is consistent with ϕ^{α}. ■

Proof. We prove by induction on the structure of the property ϕ that $\pi^i \models \phi \Leftrightarrow \lfloor\pi\rfloor_{\Lambda_{keep}}^{p(i)} \models \phi$. We have to distinguish the base case, namely considering properties on actions, and ten inductive cases on ϕ. The base case of the proof on ϕ itself is divided into for base cases for $\top$, $\bot$, inclusion and exclusion of actions w.r.t. a particular set of actions, and two inductive cases for the conjunction and disjunction of action formulae.

Base Case: Assume, $\lfloor\pi\rfloor_{\Lambda_{keep}}[i] = \lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)]$ for any $i \in \mathbb{N} \setminus \{0\}$. There are six subcases to be considered. The cases a) to d) are base cases, while e) and f) are inductive cases on ζ.

Base Cases:

a) $\pi[i] \models \top \Leftrightarrow \lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] \models \top$ is trivially true, no matter whether $\lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] = \pi[i] \in \Lambda_{keep}$ or $\lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)], \pi[i] \notin \Lambda_{keep}$.

b) The same holds trivially for $\pi[i] \models \bot \Leftrightarrow \lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] \models \bot$.

c) To prove $\pi[i] \models \{a(x)|expr(x)\} \Leftrightarrow \lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] \models \{a(x)|expr(x)\}$, we have to distinguish the two cases, whether $\pi[i] \in \Lambda_{keep}$ or $\pi[i] \notin \Lambda_{keep}$. If $\pi[i] \in \Lambda_{keep}$, then $\lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] = \pi[i]$ and the above trivially holds. If $\pi[i] \notin \Lambda_{keep}$, then $\lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] = \tau \notin \Lambda_{keep}$. In this case, $\pi[i] \not\models \{a(x)|expr(x)\}$ and nor does $\lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)]$.

d) To prove $\pi[i] \models \neg\{a(x)|expr(x)\} \Leftrightarrow \lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] \models \neg\{a(x)|expr(x)\}$, we again have to distinguish the two cases, whether $\pi[i] \in \Lambda_{keep}$ or $\pi[i] \notin \Lambda_{keep}$. If $\pi[i] \in \Lambda_{keep}$, then $\lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] = \pi[i]$ and the above trivially holds. If $\pi[i] \notin \Lambda_{keep}$, then $\lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] = \tau \notin \Lambda_{keep}$. In this case, tautologically $\pi[i] \not\models \{a(x)|expr(x)\} \Leftrightarrow \pi[i] \models \neg\{a(x)|expr(x)\}$ and, thus $\lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] \models \neg\{a(x)|expr(x)\}$.

Inductive steps:

e) This case is an inductive step.

$$\begin{aligned} &\pi[i] \models \zeta_1 \wedge \zeta_2 \\ \Leftrightarrow &\pi[i] \models \zeta_1 \wedge \pi[i] \models \zeta_2 \end{aligned}$$

By induction hypothesis:

$$\begin{aligned} \Leftrightarrow &\lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] \models \zeta_1 \wedge \lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] \models \zeta_2 \\ \Leftrightarrow &\lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] \models \zeta_1 \wedge \zeta_2 \end{aligned}$$

f) This case is also an inductive step.

$$\begin{aligned} &\pi[1] \models \zeta_1 \vee \zeta_2 \\ \Leftrightarrow &\pi[1] \models \zeta_1 \vee \pi[1] \models \zeta_2 \end{aligned}$$

By induction hypothesis:

$$\begin{aligned} \Leftrightarrow &\lfloor\pi\rfloor_{\Lambda_{keep}}[p(1)] \models \zeta_1 \vee \lfloor\pi\rfloor_{\Lambda_{keep}}[p(1)] \models \zeta_2 \\ \Leftrightarrow &\lfloor\pi\rfloor_{\Lambda_{keep}}[p(1)] \models \zeta_1 \vee \zeta_2 \end{aligned}$$

Inductive step for $\Box$ a) Assume, that $\pi \models \Box\phi$. Then for all $i \in \mathbb{N} \setminus \{0\} : \pi[i] \models \phi$. By induction hypothesis, then also for all $i \in \mathbb{N} \setminus \{0\} : \lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] \models \phi$ and thus $\lfloor\pi\rfloor_{\Lambda_{keep}} \models \Box\phi$.

Inductive step for $\Box$ b) Assume, that $\pi \not\models \Box\phi$. Then there exists an $i \in \mathbb{N} \setminus \{0\}$ such that $\pi[i] \not\models \phi$. By induction hypothesis, then also $\lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] \not\models \phi$ and thus $\lfloor\pi\rfloor_{\Lambda_{keep}} \not\models \Box\phi$.

Inductive step for $\Diamond$ a) Assume, that $\pi \models \Diamond\phi$. Then there exists an $i \in \mathbb{N} \setminus \{0\}$ such that $\pi[i] \models \phi$. By induction hypothesis, then also $\lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] \models \phi$ and thus $\lfloor\pi\rfloor_{\Lambda_{keep}} \models \Diamond\phi$.

Inductive step for $\Diamond$ b) Assume, that $\pi \not\models \Diamond\phi$. Then for all $i \in \mathbb{N} \setminus \{0\} : \pi[i] \not\models \phi$. By induction hypothesis, then also for all $i \in \mathbb{N} \setminus \{0\} : \lfloor\pi\rfloor_{\Lambda_{keep}}[p(i)] \not\models \phi$ and thus $\lfloor\pi\rfloor_{\Lambda_{keep}} \not\models \Diamond\phi$.

Inductive step for $\wedge$: Assume, that $\pi \models \phi \wedge \psi$. Then:

$$\begin{aligned} &\pi^i \models \phi \wedge \psi \\ \Leftrightarrow &\pi^i \models \phi \wedge \pi^i \models \psi \end{aligned}$$

By induction hypothesis:

$$\begin{aligned} \Leftrightarrow &\lfloor\pi\rfloor^{p(i)}_{\Lambda_{keep}} \models \phi \wedge \lfloor\pi\rfloor^{p(i)}_{\Lambda_{keep}} \models \psi \\ \Leftrightarrow &\lfloor\pi\rfloor^{p(i)}_{\Lambda_{keep}} \models \phi \wedge \psi \end{aligned}$$

Inductive step for $\vee$: Assume, that $\pi \models \phi \vee \psi$. Then:

$$\begin{aligned} &\pi^i \models \phi \vee \psi \\ \Leftrightarrow &\pi^i \models \phi \vee \pi^i \models \psi \end{aligned}$$

By induction hypothesis:

$$\begin{aligned} \Leftrightarrow &\lfloor\pi\rfloor^{p(i)}_{\Lambda_{keep}} \models \phi \vee \lfloor\pi\rfloor^{p(i)}_{\Lambda_{keep}} \models \psi \\ \Leftrightarrow &\lfloor\pi\rfloor^{p(i)}_{\Lambda_{keep}} \models \phi \vee \psi \end{aligned}$$

Inductive step for U a) Assume, that $\pi \models \phi U \psi$. This means, that there exists a $k \in \mathbb{N} \setminus \{0\}$ such that for all $0 < i < k : \pi^i \models \phi$ and $\pi^k \models \psi$.

By induction hypothesis, for all $0 < j < p(k) : \lfloor\pi\rfloor^{j}_{\Lambda_{keep}} \models \phi$ and $\lfloor\pi\rfloor^{p(k)}_{\Lambda_{keep}} \models \psi$. From this, we immediately derive that also $\lfloor\pi\rfloor_{\Lambda_{keep}} \models \phi U \psi$.

Inductive step for U b) Assume, that $\pi \not\models \phi U \psi$. This means, that either there exists a $k \in \mathbb{N} \setminus \{0\}$ such that there exists an i, $0 < i < k$, such that $\pi^i \not\models \phi$ and $\pi^k \models \psi$, or for all $k \in \mathbb{N} \setminus \{0\} : \pi^k \models \psi$. By induction hypothesis and due to the surjectivity of p_π, we derive that there thus also exists an

$p_\pi(i)$, $0 < p_\pi(i) < p_\pi(k)$, such that $\lfloor\pi\rfloor_{\Lambda_{keep}}^{p(i)} \not\models \phi$ and $\lfloor\pi\rfloor_{\Lambda_{keep}}^{p(k)} \models \psi$, or for all $p(k) \in \mathbb{N} \setminus \{0\} : \lfloor\pi\rfloor_{\Lambda_{keep}}^{p(k)} \models \psi$. From this, we can derive that also $\lfloor\pi\rfloor_{\Lambda_{keep}} \not\models \phi \mathbf{U} \psi$.

Inductive step for R a) Assume, that $\pi \models \phi \mathbf{R} \psi$. This means, that for all $k \in \mathbb{N} \setminus \{0\}$ holds that if for all $i \in \mathbb{N} \setminus \{0\}$ with $i < k$ we have if $\pi^i \not\models \phi$, then $\pi^i \models \psi$.

By induction hypothesis, for all $k \in \mathbb{N} \setminus \{0\}$ and for all $i \in \mathbb{N} \setminus \{0\}$ with $i < k$ we have if $\lfloor\pi\rfloor_{\Lambda_{keep}}^{i} \not\models \phi$, then $\lfloor\pi\rfloor_{\Lambda_{keep}}^{i} \models \psi$. From this, we immediately derive that also $\lfloor\pi\rfloor_{\Lambda_{keep}} \models \phi \mathbf{R} \psi$.

Inductive step for R b) Assume, that $\pi \not\models \phi \mathbf{R} \psi$. This means, that there exists a $k \in \mathbb{N} \setminus \{0\}$ with $\pi^k \not\models \phi \wedge \pi^k \not\models \psi$.

By induction hypothesis and due to the surjectivity of p_π, we derive that there thus also exists an $p_\pi(i)$, such that $\lfloor\pi\rfloor_{\Lambda_{keep}}^{p(i)} \not\models \phi$ and $\lfloor\pi\rfloor_{\Lambda_{keep}}^{p(i)} \not\models \psi$. From this, we can derive that also $\lfloor\pi\rfloor_{\Lambda_{keep}} \not\models \phi \mathbf{R} \psi$.

□

The next definition introduces a *projection relation*, which defines two traces as being equivalent, if they only differ in their irrelevant parts, i.e. in the relaxed actions. In this case, their projections to Λ_{keep} are equal.

Definition 7.27 (Projection Relation). Two traces π_1, π_2 are equivalent under projection relation $\pi_1 \sim_p \pi_2$, if and only if $\lfloor\pi_1\rfloor_{\Lambda_{keep}} = \lfloor\pi_2\rfloor_{\Lambda_{keep}}$. ■

This definition forms the second invariance definition for properties. Properties are considered to be invariant, if two traces are in a projection relation with each other, either both traces satisfy the property or they both do not.

Definition 7.28 (Invariance of Properties (2)). A property ϕ is invariant under projection relation $\sim_p$ if and only if the following holds: $\forall \pi_1, \pi_2 : \pi_1 \sim_p \pi_2 \Rightarrow (\pi_1 \models \phi \Leftrightarrow \pi_2 \models \phi)$. ■

Lemma 7.29. Any eALTL property is invariant for an arbitrary pair of traces π_1, π_2 with $\pi_1 \sim_p \pi_2$. ■

Proof. As defined in Definition 7.27, $\pi_1 \sim_p \pi_2 \Leftrightarrow \lfloor\pi_1\rfloor_{\Lambda_{keep}} = \lfloor\pi_2\rfloor_{\Lambda_{keep}}$. As it has been proven in Lemma 7.26, any eALTL property is invariant under projection p_π from π to $\lfloor\pi\rfloor_{\Lambda_{keep}}$.
By Lemma 7.26, $\pi_1 \models \phi \Leftrightarrow \lfloor\pi_1\rfloor_{\Lambda_{keep}} \models \phi$. Since $\lfloor\pi_1\rfloor_{\Lambda_{keep}} = \lfloor\pi_2\rfloor_{\Lambda_{keep}}$, we derive that also $\lfloor\pi_1\rfloor_{\Lambda_{keep}} \models \phi \Leftrightarrow \lfloor\pi_2\rfloor_{\Lambda_{keep}} \models \phi$. By Lemma 7.26, this also means that $\lfloor\pi_2\rfloor_{\Lambda_{keep}} \models \phi \Leftrightarrow \pi_2 \models \phi$. From this, we conclude, that also $\pi_1 \models \phi \Leftrightarrow \pi_2 \models \phi$. □

Lemma 7.30. For any pair of traces $\pi_1, \pi_2 \in [\![\mathfrak{V}]\!]_{traces}$ we have: $\pi_1 \sim_p \pi_2$. ■

Proof. Having in mind the construction of the violation pattern $\mathfrak{V}$, it is trivial to prove this lemma. Every trace $\pi \in [\![\mathfrak{V}]\!]_{traces}$ consists of steps according to transitions $\sigma \xrightarrow{\lambda}$

$\hat{\sigma} \in \Delta$ with $\sigma \neq \hat{\sigma}$ or those accordant to self loops $\sigma \xrightarrow{\lambda} \hat{\sigma} \in \Delta$. Infinite self loops in the cyclic state are not possible by construction. Furthermore, by construction of the violation pattern, any self loop in $\mathfrak{V}$ is preceded by a step $\sigma \xrightarrow{\lambda} \hat{\sigma}$, $\sigma \neq \hat{\sigma}$, with $\lambda \notin \Lambda_{keep}$. This means, for any trace $\pi \in [\![\mathfrak{V}]\!]_{traces}$, holds:

- $\forall i \in \mathbb{N} \setminus \{0\}, \pi[i] \notin \Lambda_{keep} \exists p(i) \in \mathbb{N} \setminus \{0\} : \pi'[p(i)] = \tau$
- $\forall j \in \mathbb{N} \setminus \{0\}, \pi[j] \in \Lambda_{keep} \exists p(j) \in \mathbb{N} \setminus \{0\} : \pi'[p(j)] = \pi[j]$

In applying this projection, the trace π' is the shortest trace through $\mathfrak{V}$ skipping all self loops. Modulo actions $\lambda \in \Lambda \setminus \Lambda_{keep}$, by construction there is only one such trace in $\mathfrak{V}$. For this reason, any trace $\pi \in [\![\mathfrak{V}]\!]_{traces}$ will be projected on π' and thus the lemma holds. □

Lemma 7.31. Let Λ_{keep} be consistent with ϕ^α, let χ^α be a counterexample for ϕ^α, and $\mathfrak{V}$ be a violation pattern generated from χ^α and Λ_{keep}. Every trace $\pi^\alpha \in [\![\mathfrak{V}]\!]_{traces}$ satisfies: $\pi^\alpha \not\models \phi^\alpha$. ■

Proof. It trivially holds that $\chi^\alpha \not\models \phi$. As we have shown in Lemma 7.26, any eALTL property is invariant under the projection p_π of trace π to trace $\lfloor \pi \rfloor_{\Lambda_{keep}}$. As has been shown in Lemma 7.30, for any two traces $\pi_1, \pi_2 \in [\![\mathfrak{V}]\!]_{traces}$ holds: $\pi_1 \sim_p \pi_2$. Furthermore, we have shown in Lemma 7.29, that any eALTL property is invariant under the equivalence $\pi_1 \sim_p \pi_2$. From this, we can derive that every trace $\pi^\alpha \in [\![\mathfrak{V}]\!]_{traces}$ satisfies: $\pi^\alpha \not\models \phi^\alpha$. □

7.5.2 Looking for a concrete counterexample

After we have constructed the violation pattern $\mathfrak{V}$, we check whether there is a concrete counterexample $\chi = \chi_p \chi_s^\omega$, such that the corresponding abstract counterexample $\chi^\alpha \in [\![\mathfrak{V}]\!]_{traces}$.

For infinite counterexamples we need to check that some state of χ_s corresponds to σ_{cyclic}. We employ constraint solving (Marriott and Stuckey, 1998) to find a concrete counterexample, which allows us to check this condition for infinite (but cyclic) traces, and also for certain infinite and parameterized systems.

To find a concrete trace matching the violation pattern $\mathfrak{V}$, we transform the specification of the concrete system and the violation pattern into a CLP, and formulate a query to find such a trace. This transformation is similar to the one described in Section 3.3. Note that for a concrete system with an infinite state space, it is possible that the constraint solver will not terminate. Moreover, it is possible that the only traces that match the violation pattern are spiral traces, not cyclic ones (i.e. we do have a loop with respect to control locations, but some variable is infinitely growing) and we will not be able to find them.

Transformation of the edges of the violation pattern $\mathfrak{V} = (\Sigma, \Lambda, \Delta, \sigma_{init}, \sigma_{cyclic})$ into the rules of the CLP $\mathfrak{P}_\mathfrak{V}$ is defined in Table 7.1. Here, we abbreviate $(\ell, \overline{Var})$ by $\vec{x}$ and $(\hat{\ell}, \overline{Var'})$ by $\vec{x}'$. Intuitively, given a step of $\mathfrak{V}$, a rule of $\mathfrak{P}_\mathfrak{V}$ checks whether the

$$\text{VII-a}\ \frac{\sigma \xrightarrow{!s(\nu)} \hat{\sigma} \vee \sigma \xrightarrow{?s(\nu)} \hat{\sigma} \vee \sigma \xrightarrow{\tau} \hat{\sigma} \quad \sigma \neq \sigma_{\text{cyclic}} \quad \hat{\sigma} \neq \sigma_{\text{cyclic}}}{\begin{array}{c}\sigma(state(\vec{x}), \bar{C}, [s(y) \mid \bar{\pi}]) \leftarrow s(state(\vec{x}), state(\vec{x}'), param(y)) \wedge \\ \nu = \alpha(y) \wedge \hat{\sigma}(state(\vec{x}'), \bar{C}, \bar{\pi})\end{array}}$$

$$\text{VII-b}\ \frac{\sigma \xrightarrow{\tau} \hat{\sigma} \quad \sigma \neq \sigma_{\text{cyclic}} \quad \hat{\sigma} = \sigma_{\text{cyclic}}}{\sigma(state(\vec{x}), \bar{C}, \bar{\pi}) \leftarrow \hat{\sigma}(state(\vec{x}), \bar{C}, \bar{\pi})}$$

$$\text{VII-c}\ \frac{\sigma \xrightarrow{!s(\nu)} \hat{\sigma} \vee \sigma \xrightarrow{?s(\nu)} \hat{\sigma} \vee \sigma \xrightarrow{\tau} \hat{\sigma} \quad \sigma = \sigma_{\text{cyclic}} \quad \vec{x} \in \bar{C}}{\sigma(state(\vec{x}), \bar{C}, []) \leftarrow \vec{x} \in \bar{C}}$$

$$\text{VII-d}\ \frac{\sigma \xrightarrow{!s(\nu)} \hat{\sigma} \vee \sigma \xrightarrow{?s(\nu)} \hat{\sigma} \vee \sigma \xrightarrow{\tau} \hat{\sigma} \quad \sigma = \sigma_{\text{cyclic}} \quad \vec{x} \notin \bar{C}}{\begin{array}{c}\sigma(state(\vec{x}), \bar{C}, [s(y) \mid \bar{\pi}]) \leftarrow s(state(\vec{x}), state(\vec{x}'), param(y)) \wedge \\ \nu = \alpha(y) \wedge \hat{\sigma}(state(\vec{x}'), [\vec{x} \mid \bar{C}], \bar{\pi})\end{array}}$$

Table 7.1: From violation pattern $\mathfrak{V}$ to CLP $\mathfrak{P}_{\mathfrak{V}}$

concrete system may make this step. The rules also take into account the information about the cyclic state and the data abstraction.

The rules in Table 7.1 transform the steps of a violation pattern into rules of the form: $\rho \leftarrow \xi \wedge g_\alpha \wedge \nu$. ρ is a user-defined constraint of the form $\sigma(state(\vec{x}), \bar{C}, \bar{\pi}')$ specifying the source state $state(\vec{x})$ of the concrete system, and a set $\bar{C}$ of states, which are *possibly* on a cycle. This set is accumulatively constructed, and it contains concrete candidate cyclic states that match σ_{cyclic} in the violation pattern. The third parameter, $\bar{\pi}'$, contains the trace that is visited while examining $\mathfrak{V}$ starting from $\hat{\sigma}$ and the action visited in the actual step.

ξ is a user-defined constraint of the form $s(state(\vec{x}), state(\vec{x}'), param(y))$ as defined above. It represents a step on which the concrete system and the violation pattern can potentially synchronize.

The guard g_α checks whether the data parameters of the concrete action are a concretization of the data parameters of the abstract action.

Finally, ν determines whether and how the violation pattern has to be examined further. We will explain this in more detail shortly. Simplified, ν stops the further examination of $\mathfrak{V}$, if we have reached the cyclic state of $\mathfrak{V}$. Otherwise, it decides that the next step in $\mathfrak{V}$ will be taken and sets the parameters accordingly.

We will now describe the rules in more detail. Rule VII-a of Table 7.1 transforms steps of the violation pattern whose actual state σ and target state $\hat{\sigma}$ are not the beginning of the cycle base. The step specified by a constraint $s(state(\vec{x}), state(\vec{x}'), param(y))$ changes the state to $\hat{\sigma}$ in the violation pattern and to $state(\vec{x}')$ in the concrete system. That is captured by the constraint $\hat{\sigma}(state(\vec{x}'), \bar{C}, \bar{\pi})$ in ρ. The constraint is satisfied only if both the violation pattern and the concrete system can make the specified

step, and the action labeling the step of the concrete system satisfies the constraint $\nu = \alpha(y)$. When doing the next examination step, $\bar{C}$ is left unchanged. $\bar{\pi}$ is an output parameter, which contains the trace stub visited from $\hat{\sigma}$ to the cycle in the trace under examination. When the recursion ascends after termination of constraint solving, the actual event s together with a concretization y of its parameter ν, is added to the examination trace $\bar{\pi}$.

Rule VII-b transforms the τ-steps ending in the cyclic state σ_{cyclic} to an empty step from σ to $\hat{\sigma}$. The rule does not refer to any action steps from $\mathfrak{S}$, since this τ-step only appears in the violation pattern; the system under investigation stays completely silent.

Rule VII-c transforms those steps of the violation pattern that start from a state corresponding to the beginning of the cycle. If the actual corresponding state in the system is found in $\bar{C}$, the state is cyclic and has been visited already earlier during the examination. In this case, examination ends successfully. If the state is not yet in $\bar{C}$, it is *potentially cyclic* and treated by Rule VII-d. In this case, the step is treated like in Rule VII-a, just that the actual state of the system is added to $\bar{C}$. Logging potentially cyclic states and examining the violation pattern further allows us not only to detect obvious cycles, i.e. cycles in the system which are also immediately visible in the violation pattern, but also detect those cycles, where the system spirals before entering a real cycle. In this case, the system first runs through a cycle with respect to the location, but differing in the data part of the system state, before finally returning to a previously visited state. In such a case, the cyclic state of the violation pattern is visited more than once.

The CLP $\mathfrak{P}_{\mathfrak{V}}$, together with the CLP $\mathfrak{P}_{\mathfrak{S}}$, forms the constraint program. In order to check whether we can find a concrete counterexample matching the violation pattern, we transform the pair of the initial state of the violation pattern and the initial state of the concrete system into the query $q_{init} := \sigma_{init}(state(\vec{x}_{init}), [], B)$ (initial state without any potentially cyclic states and with a yet uninstantiated variable B for the counterexample trace) and ask a constraint solver, whether it finds a solution in the constraint program formed by $\mathfrak{P}_{\mathfrak{S}}$ and $\mathfrak{P}_{\mathfrak{V}}$. If yes, it provides us a counterexample as a list of actions and violation pattern states, which has been collected over the examination of $\mathfrak{V}$. If constraint solving does not find a solution, we cannot give a conclusive answer and have to use e.g. abstraction refinement techniques to find out, whether the property holds on the concrete system.

Lemma 7.32. For all $\vec{z} \in \bar{C}$ holds that, if $\sigma(state(\vec{x}), \bar{C}, \pi)$ is invoked, then $\vec{z} \nrightarrow \vec{x}$. ■

Proof. We will prove the lemma by induction over the deduction steps for the query to the CLP.

Base case: This case is given by the initial invocation of $q_{init} := \sigma_{init}(state(\vec{x}_{init}), [], B)$. In this case, $\bar{C} = \emptyset$, such that the hypothesis trivially holds for the first step.

Inductive step: For the inductive step, we assume, that $\vec{z} \in \bar{C} \wedge \vec{z} \nrightarrow \vec{x}$. We now have to distinguish four cases according to the rules from Table 7.1:

Case 1: By induction hypothesis, we have a trace $\vec{z} \twoheadrightarrow \vec{x}$ with $\vec{z} \in \bar{C}$. A successful invocation of Rule VII-a induces, that there is also a step $\vec{x} \xrightarrow{s(y)} \vec{x}'$, since the invocation of $s(state(\vec{x}), state(\vec{x}'), param(y))$ succeeds. From this, we can derive that $\vec{z} \twoheadrightarrow \vec{x} \rightarrow \vec{x}'$ and thus $\vec{z} \twoheadrightarrow \vec{x}'$.

Case 2: This case trivially holds due to our induction hypothesis.

Case 3: This case also trivially holds due to our induction hypothesis with a looping trace $\vec{z} \rightarrow \vec{z}$.

Case 4: By induction hypothesis, we have a trace $\vec{z} \twoheadrightarrow \vec{x}$. For this case, we have to consider two subcases, namely $\vec{z} \in \bar{C}$ and $\vec{z} \notin \bar{C} \wedge \vec{z} \in \{\vec{z}\} \cup \bar{C}$. For the first subcase, the same holds as for Case 1 of the inductive step in this proof with $\vec{x} \neq \vec{z}$. For the second subcase, we have a step $\vec{z} \xrightarrow{s(y)} \vec{x}'$ with $\vec{z} \in \{\vec{z}\} \cup \bar{C}$ and construct the further trace by induction.

□

Lemma 7.33. If the query q_{init} to the CLP $\mathfrak{P}_{\mathfrak{V}}$ holds for some trace π, then $\pi \in [\![\mathfrak{M}]\!]_{traces}$, $\alpha(\pi) \in [\![\mathfrak{V}]\!]_{traces}$ and $\pi = \pi_p \pi_s^\omega$. ■

Proof. We prove the lemma inductively over the derivative steps of the constraint solver on the CLP $\mathfrak{P}$. We have to show that:

$$\exists \sigma^{\mathfrak{M}}_{cyclic} \in \Sigma^{\mathfrak{M}}. \exists \pi_p \pi_s \in [\![\mathfrak{M}]\!]_{traces} : \sigma_{init} \overset{\pi_p}{\twoheadrightarrow} \sigma^{\mathfrak{M}}_{cyclic} \overset{\pi_s}{\twoheadrightarrow} \sigma^{\mathfrak{M}}_{cyclic}$$

Furthermore, we have to show that the abstraction of this trace π is in $[\![\mathfrak{V}]\!]_{traces}$.

Base case: Assume, the query $\sigma(state(\vec{x}_{init}), [], \pi)$ terminates with trace $\pi = \pi_p \pi_s$. Then, we have at the end of recursion: $\vec{x} \in \bar{C}$. From that, we can derive that (Rule VII-c from Table 7.1):

$$\text{End of recursion} \frac{\sigma(state(\vec{x}), \overbrace{[\ldots, \vec{x}, \ldots]}^{\bar{C}}, []) \quad \sigma \xrightarrow{\iota} \sigma' \quad \sigma'' = \sigma_{cyclic}}{\vec{x} \in \bar{C}}$$

In this case, we have, as has been proven for Lemma 7.32, a trace $\vec{x} \twoheadrightarrow \vec{x}$ with $\sigma^{\mathfrak{M}}_{cyclic} = \vec{x}$. Since the complete trace $\pi_p \pi_s$ starts in the initial state σ_{init} (by the query), we thus have $\sigma_{init} \overset{\pi_p}{\twoheadrightarrow} \sigma^{\mathfrak{M}}_{cyclic} \overset{\pi_s}{\twoheadrightarrow} \sigma^{\mathfrak{M}}_{cyclic}$. Since the remaining empty trace does not contain any steps, it trivially holds, that its abstraction is in $[\![\mathfrak{V}]\!]_{traces}$.

Inductive step: For the inductive step, we have to consider two possible variants of steps, namely those starting in a potentially cyclic step and those starting in an arbitrary non-cyclic state (Rules VII-a and VII-d from Table 7.1):

$$\text{Var. 1}\ \frac{\sigma(state(\vec{x}''), \bar{C}, [\iota|\bar{\pi}]) \quad \sigma'' \xrightarrow{\iota} \sigma' \quad \sigma'' \neq \sigma_{cyclic}}{\frac{\iota(state(\vec{x}''), state(\vec{x}'), param(y)) \wedge \nu = \alpha(y) \wedge \sigma(\vec{x}', \bar{C}, \bar{\pi})}{g \wedge \nu = \alpha(y) \wedge \sigma(\vec{x}', \bar{C}, \bar{\pi})}}$$

$$\text{Var. 2}\ \frac{\sigma(state(\vec{x}''), \bar{C}, [\iota|\bar{\pi}]) \quad \sigma'' \xrightarrow{\tau;\iota} \sigma' \quad \sigma'' = \sigma_{cyclic}}{\frac{\iota(state(\vec{x}''), state(\vec{x}'), param(y)) \wedge \nu = \alpha(y) \wedge \sigma(\vec{x}', [\vec{x}''|\bar{C}], \bar{\pi})}{g \wedge \nu = \alpha(y) \wedge \sigma(\vec{x}', [\vec{x}''|\bar{C}], \bar{\pi})}}$$

In case of the τ-step prior to the cyclic state σ_{cyclic} in the violation pattern, an invocation to Rule VII-b of the CLP appears. This invocation does not affect the validity of a found solution trace π for the violation pattern.

In case of an arbitrary step ι, assume, the algorithm successfully terminates for a trace $\bar{\pi}$. Then, the algorithm will also successfully terminate for trace $\bar{\pi}' = [\iota|\bar{\pi}]$ with either *Variant 1* or *Variant 2* holding.

By induction hypothesis, $\bar{\pi}^\alpha \in [\![\mathfrak{V}]\!]_{traces}$. The event ι has the abstract parameter $\nu = \alpha(y)$, which is an abstraction of an appropriate concrete action parameter y. For this reason, also $\bar{\pi}'^\alpha \in [\![\mathfrak{V}]\!]_{traces}$.

□

7.5.3 Correctness of the Framework

In this section, we argue the correctness of the framework, which has been worked out in the previous two subsections.

Theorem 7.34. Let $\alpha = \langle h_s, h_a \rangle$ be an abstraction consistent with eALTL-property ϕ. Let LTSs $\mathfrak{M}$ and $\mathfrak{M}^\alpha$ be given, such that $\mathfrak{M} \subseteq_\alpha \mathfrak{M}^\alpha$. Furthermore, assume that the counterexample $\chi^\alpha \in [\![\mathfrak{M}^\alpha]\!]_{traces}$ and $\chi^\alpha \not\models \phi^\alpha$. Let $\mathfrak{V}$ be a violation pattern built from χ^α and a consistent Λ_{keep} by Algorithm 7.1. Let π be a trace for which q_{init} holds, according to the constraint solving procedure defined in Subsection 7.5.2. Then π is a counterexample: $\pi \in [\![\mathfrak{M}]\!]_{traces}$ and $\pi \not\models \phi$.

Proof. By Lemma 7.33, $\pi \in [\![\mathfrak{M}]\!]_{traces}$ and $\alpha(\pi) \in [\![\mathfrak{V}]\!]_{traces}$. By Lemma 7.31, $\alpha(\pi) \not\models \phi^\alpha$. By Lemma 7.17, as α is a precise abstraction, we have $\pi \not\models \phi$. □

7.6 A Case Study: PAR

To check the applicability of our framework, we performed a number of verification experiments with μCRL specifications. For our experiments, we took a mutant of the Positive Acknowledgement Retransmission Protocol (PAR; Tanenbaum, 1981) as a case study. The usual scenario for PAR includes a sender, a receiver, a message channel and an acknowledgment channel as can be seen in Figure 7.10.

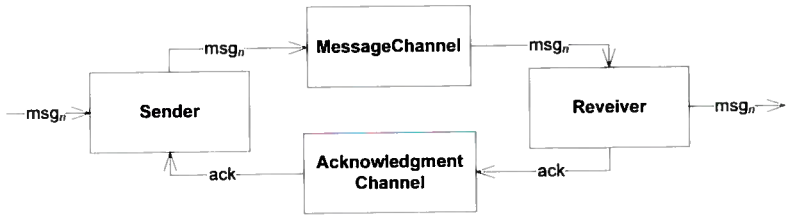

Figure 7.10: Collaboration in PAR

The sender receives a frame from the upper layer, i.e. from its environment, sends it to the receiver via the message channel, the receiver delivers the frame to the upper layer and sends a positive acknowledgment via the acknowledgment channel to the sender. PAR depends on timers. The channels in the system delay the delivery of messages and can, moreover, lose or corrupt messages. In the mutant we used for our experiments, we have chosen too short timers, in order to provoke a bug in the system.

When the receiver has delivered the message to the upper layer, it sends an acknowledgment to the sender. After the positive acknowledgment has been received, the sender becomes ready to send a subsequent message. The sender handles lost frames by timing out. If the sender times out, it re-sends the message.

We tried to verify that for any setting of the sender timer exceeding some value k, all messages sent by the upper layer to the sender are eventually received by the upper layer from the receiver. To prove that the property holds for any setting of the sender timer exceeding k, we applied the timer abstraction described in the introduction of this chapter to the sender timer. The property was not satisfied on the abstract system (the chosen k was less than the sum of the channel delays) and we obtained a counterexample.

Figure 7.11 illustrates the idea of this well-known erroneous scenario for PAR. The sender times out while the acknowledgment is still on the way. The sender sends a duplicate, then receives the acknowledgment and believes that this is the acknowledgment for the duplicate. The sender sends the next message, which gets lost. However, the sender receives the acknowledgment for the duplicate, which it believes to be the acknowledgment for the last message. Thus the sender does not retransmit the lost message and the protocol fails. To avoid this erroneous behavior, the timeout interval must be long enough to prevent a premature timeout, which means that the timeout interval should be larger than the sum of delays on the message channel, the acknowledgment channel and the receiver (Tanenbaum, 1981).

The abstract counterexample was not reproducible on the concrete system, since the number of tick steps from a setting of the sender timer till its expiration varied along the trace, due to the use of the abstraction. We transformed the counterexample into the violation pattern from Figure 7.12, by relaxing the actions on the sender timer as influenced by the abstraction. The violation pattern basically says that after losing

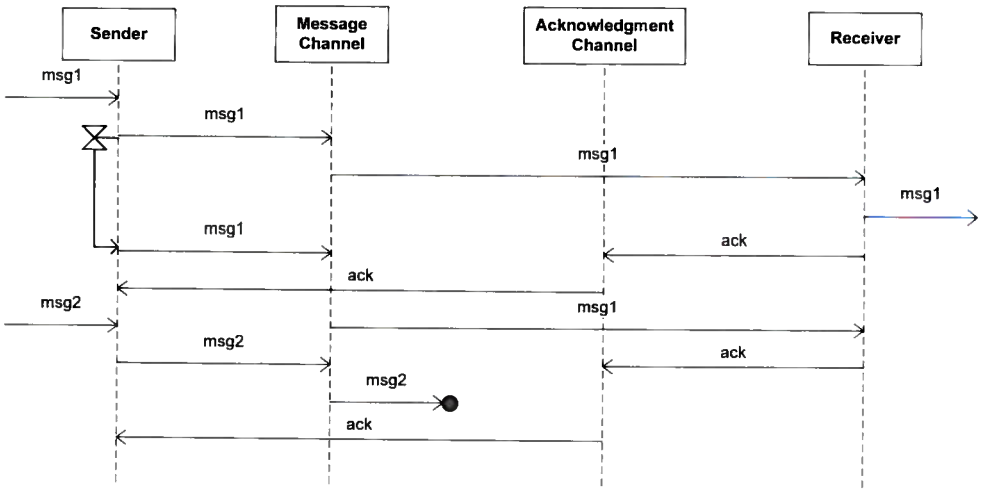

Figure 7.11: Counterexample for the mutant of PAR

message msg2, the receiver will reject delivery of message msg3. Further the system gets into the loop where again and again message msg3 is delivered first with the sequence bit T and then with the sequence bit F. Thus, the message msg2 has never been delivered.

The specification of the system was transformed from μCRL into a Prolog CLP, while the violation pattern was immediately formulated as a set of Prolog rules according to our theory (Definitions 7.22, 7.23 and Figure 7.9). The constraint solver was then able to find a concrete counterexample for our property.

In the realization of the CLP for our experiment, we do not keep track of cyclic states by building up a set $\bar{C}$, but by dynamically adding rules to the CLP using the Prolog directive $assert/1$. Therefore, we introduce a new dynamic rule $cyclic\_state/1$. In the rule for the cyclic state of the violation pattern, we insert instances of this rule using $assert(cyclic\_state(state(\vec{x})))$, to mark the cyclic state before following the transition from that state. For the final state of $\pi_p\pi_s$, we add two rules: The first rule checks, whether a state in $\bar{C}$ was reached by invoking $cyclic\_state(state(\vec{x}))$. If this rule holds, the state was reached, the trace $\bar{\pi}$ is printed out and constraint solving terminates. If this rule fails, the second rule invokes the rule for the cyclic state.

7.7 Related work

First, we compare our method with the CEGAR approach (Counter-Example-Guided Abstraction Refinement) by Clarke et al. (2003) and Lakhnech et al. (2001), which has recently been extended to state- and event-based software by the ComFoRT frame-

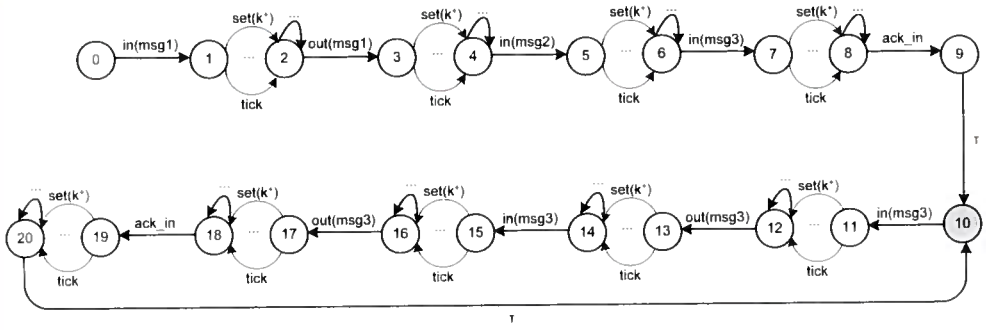

Figure 7.12: Violation pattern for PAR

work (Chaki et al., 2005). In both methods, abstractions preserve properties in one direction only: if the abstract system satisfies the property, so does the concrete system; a counterexample may however be a real one or a false negative. In the CEGAR method, the abstraction is refined based on abstract counterexamples, and model checking is iteratively applied to the refined abstractions of the system. Our method is to generalize false negatives and then to find violations in the concrete specification, which are similar to the original false negative. Note that in principle both methods can be combined: given a false negative, one could search for a concrete violation using our method. If it is found, the CEGAR loop can be terminated early. If no concrete counterexample is found, one can proceed by refining the abstraction as in the CEGAR approach and iterate the verification process.

For counterexamples that have been produced when model checking the abstract model, it has to be determined whether they represent real system defects. The problem of automating this analysis has been addressed by Păsăreanu et al. (2001). For this purpose, the authors propose two techniques: model checking on choice-free paths and abstract counterexample guided concrete simulation. An approach based on test generation is proposed by Pace et al. (2004) for searching for concrete instances of abstract counterexamples. Only counterexamples for safety properties are addressed by those approaches, i.e. it works only for finite counterexamples, while we deal with infinite traces. Unlike these approaches, we look for a concrete trace that does not match a counterexample itself, but a violation pattern that has been generated from it.

The approaches of Grumberg et al. (2005) and Păsăreanu et al. (2005) are orthogonal to ours, because there model checking methods are proposed that rely on a refinement of an *underapproximation* of the system behavior. These methods are aimed at the falsification of a desired property and apply a refinement when no counterexample is found. In contrast, we aim at proving the property and, in case we do not succeed, try to find a concrete counterexample.

In SLAM (Ball et al., 2004), C programs with a set of predicates are abstracted into

a boolean program. If the Boolean program contains an error path and this path is a feasible execution path in the original C, then the process has found a potential error. If this path is not feasible in the C program, the boolean program is refined to eliminate this false negative. Again as with Păsăreanu et al. (2001), one searches for an exact matching between an abstract and a concrete counterexample.

A general abstraction-refinement approach has been presented by Clarke et al. (2003). There, the abstraction is refined if the used one is not fine enough to find an abstract counterexample having an exact concrete counterpart. The algorithm presented by Clarke et al. allows the calculation of a concrete set of paths from a set of abstract ones. If the result is empty, then abstract counterexamples clearly have no counterpart in the concrete system. The method is dependent on the computability of the inverse of the used abstraction function, and also on the decidability of whether the set of traces is empty or not.

Chapter 8

Conclusion

> Den Anfang des allgemeinen Hellerwerdens draußen vor dem Fenster erlebte er noch. Dann sank sein Kopf ohne seinen Willen gänzlich nieder, und aus seinen Nüstern strömte sein letzter Atem schwach hervor.
>
> *(Franz Kafka)*

It is the task of software quality assurance to find and avoid software faults, but this is a cumbersome, expensive and also erroneous undertaking. For this reason, research has been done over the last years in order to automate this task as much as possible. In this thesis, the connection of constraint solving techniques with formal methods is investigated with the goal to find faults in models and implementations of reactive systems with data. In this chapter, we will discuss the achievements of this thesis and give some outlook for future research directions, but also for concrete improvements regarding the tooling of the theories.

Testing with Data Abstraction

Results: In Chapter 4, we proposed an approach to generate test cases combining data abstraction, enumerative test generation and constraint-solving. Given the concrete specification of an open system, the presented data abstraction allows to derive the appropriate abstract system that is finite with respect to data exchanged with its environment and thus suitable for the automatic generation of abstract test cases with enumerative tools. In order to execute the test cases, we have to instantiate them with concrete data. For data selection we make use of constraint-solving techniques: a CLP is derived from the system specification and then solved by a constraint solver. The derivation of the CLP is based on the theory from Chapter 3. Based on the solution of the CLP, the test cases can be parameterized and executed. We have proven the correctness of our approach. To corroborate the applicability of our approach, we applied it to the CEPS case study (Jürjens and Wimmel, 2001; CEPSCO, 1999, 2000).

Outlook: An interesting and necessary aspect, especially from a practical viewpoint, is the generation of test cases directly from UML specifications, as it has been proposed, for instance, by Offutt and Abdurazik (1999) as well as Briand and Labiche (2001). In Calamé et al. (2006b), we proposed a method for the generation of test cases in TTCN-3 from a subset of UML. This approach combines well with the research presented in Chapter 4 because it generates a µCRL specification from a UML model and then proceeds further with the presented data abstraction, test case and

test oracle generation. The generation of test code targeting TTCN-3 also embeds this approach into a TTCN-3-based framework of testing methodologies and tools, as it has been developed within the TT-Medal project. Existing TTCN-3 execution frameworks allow a static parameterization of test cases with test data, which has been selected prior to the test run itself. In this context it would interesting to combine TTCN-3 and constraint solving techniques to automate test parametrization using constraint solvers or even to allow a dynamic test case parameterization during the test run. The latter approach would be comparable to test execution with BAiT.

Behavior Adaptation in Testing

Results: With BAiT, we developed in Chapter 5 a toolset for the generation and execution of conformance test cases of systems with data. It can be seen as an extension of automatic test case generation by TGV (Jard and Jéron, 2005) or STG (Clarke et al., 2002). Embedded into the test generation and execution process proposed in the same chapter, it can provide a nearly fully automatic test of a system.

Automation starts at specification level. The whole process covers data abstraction, test generation with TGV (both as presented in Chapter 4) and the generation of a test oracle. After the reintroduction of variable names in the generated test cases and the generation of the proxies to the IUT, the test can be run automatically. While executing a test, BAiT adapts to unforeseen behavior of the IUT in order to prevent false positive Inconc verdicts. The toolset is extensible and allows, for instance, the introduction of new trace-search strategies based on ideas like game theory, as they have, for instance, been worked out by Nachmanson et al. (2004). It also allows the introduction of data selection algorithms like pattern-based data selection as it is supported by the tool TOBIAS (Ledru et al., 2001).

Outlook: However, we are still capable to further improve the tool BAiT. One issue here is to improve system-independency of the framework. Even though its architecture should allow a relatively easy adaptation of the framework to other platforms than Java in a procedure-based setting, its viability w.r.t. this issue is still to be tested with other communication paradigms and other platforms. The test execution unit of TTCN-3 is quite advanced here, however, paying the price of an increased complexity of the approach.

Another issue is the handling of non-trivial, i.e. non-numerical and non-enumerative datatypes. BAiT does in this point suffer from the fact, that ECLiPSe Prolog provides data ranges for data selection only for numerical and enumerative datatypes. For all other kinds of data, it only returns the first possible solution. This problem can only be solved by improving the generated test oracles using the Prolog predicate `findall` (Novello et al., 2005, Section 8.4.3) and adapting the generation of queries as well as the interpretation of constraint solving results on the side of the BAiT runtime.

The third issue is a system-inherent one: Even though BAiT is a blackbox test framework, it allows the definition and "execution" of IUT-internal actions (τ-steps). In order to do so, BAiT tries to predict the internal behavior of the IUT during the

test. If now the specification of the IUT allows logic within these internal actions (branches or loops), this prediction can get out of sync with the actual internal behavior of the IUT. For this reason, BAiT can only reduce the number of false positive Fail verdicts for such systems, but not completely avoid them.

Case Studies for Testing with BAiT

In Chapter 6, BAiT has first successfully been evaluated with an academic case, the ATM case study. An ATM as an action-based system with nondeterministic behavior, such as the expenditure of money, and data is the ideal target for testing with BAiT.

In a second case study, we evaluated BAiT on an SUT, which is not oriented to processes, but to documents: the Mozilla Gecko HTML rendering engine. This is a new application for BAiT (and also for automated conformance testing in general), since the toolset was originally designed to treat nondeterministic reactive, and thus action-based, systems with data. We modeled a fragment of the CSS box model in μCRL, implemented a wrapper for Mozilla Gecko in order to be able to apply BAiT for test execution, and designed some test cases following the BAiT approach. In a controlled experiment, we validated the applicability of BAiT to document-centered HTML rendering.

Our experiences with the design of test purposes were twofold. On the one hand the behavior-oriented part of the design was very easy. On the other hand, this means that most of the test design is induced by data and thus actually happens during the test run itself and can be computed automatically. Hereby, we encountered, that the data parameters in this case study were mainly independent of each other, so that BAiT could not show all its capabilities. Also, BAiT's behavior adaptation could easily lead to test runs, which did not terminate. In such a case, a careful configuration of the search threshold of BAiT is necessary to prevent infinite test runs while at the same time avoiding superfluous Inconc verdicts.

The feasibility study described in this thesis was successful and forms an important step towards a fully automated model-based test approach for HTML rendering engines using BAiT. Compared to static test suites, which serve the document-centered HTML rendering process, our behavior-oriented approach has the advantage of a higher flexibility w.r.t. data parameters and the tested document structure. It also better suits dynamic web page construction as it is nowadays practiced using techniques as AJAX. With a wrapper for Gecko and a behavior-oriented approach to test the IUT, it is far easier to generate a variety of different HTML test documents, which cover different aspects of the IUT. A direction for future on this topic is surely the extension of our chosen fragment of the CSS box model to the full model by the W3C (2007). However, since the model of the W3C (2007) is given in natural language rather than a formal notation, it might quite likely be incomplete, ambiguous and may contain semantic variation points. On the technical level, an extension of the case study should include such interesting issues like the treatment of exceptional situations in rendering a web page, like an overflowing layout. Furthermore, floating point datatypes should be supported by the model at least to the extent, the

ECLiPSe Prolog constraint solver supports them. This task is not really simple, since it includes amongst other things the treatment of data, which is not exact but suffers from rounding errors. In order to properly treat this kind of data, we have to consider data ranges rather than exact values when interpreting the reaction of the IUT on input during test execution.

Bug Hunting with False Negatives

For bug hunting with false negatives, we proposed in Chapter 7 a novel framework for interpreting negative verification results obtained with the help of data abstractions. Existing approaches to handling abstract counterexamples try to find an exact counterpart of the counterexample (e.g. Păsăreanu et al., 2001). When no concrete counterpart can be found, data abstraction is considered to be not fine enough and abstraction refinement is applied (e.g. Clarke et al., 2003; Brückner et al., 2007).

Results: In our framework we look for useful information in false negatives, combining the two formal methods model checking and constraint solving. Given a specification of a system and a property (formulated as an eALTL formula), we first choose and apply data abstraction to both of them and then verify the abstract property on the abstract system. If the verification results in a violation of the abstract property and the obtained counterexample has no counterpart in the concrete system, we transform the counterexample into a violation pattern, which is further used to guide the search for concrete counterexamples.

The framework allows to handle counterexamples obtained when verifying safety properties, but also counterexamples for liveness properties. Moreover, the framework can be applied for searching concrete counterexamples in parameterized and infinite state systems. Success is not always guaranteed – the violation pattern can be too strict, concrete counterexamples can have a spiral form (i.e. a loop in the specification, that does not lead back to a state fully identical to its starting state), or there could be no counterexample at all since the property just holds on the concrete system. Still, our approach can help in finding counterexamples in those cases when a data abstraction influences the order and the number of some actions, e.g. as timer and counter abstractions do. Even though we defined the framework for homomorphistic abstractions in this thesis, it seems to be possible to generalize abstraction and refinement on the basis of Galois-connections and so define a framework for bughunting with false negatives based on abstract interpretation.

Outlook: The approach to the generation of a violation pattern leaves a certain freedom in the sense that the set of actions to relax can be more/less restrictive. Tuning the violation pattern or using the expertise of system developers to pick an appropriate set of actions to relax can be potentially less costly than repeating the abstraction/refinement cycle immediately. As an alternative to such a manual approach, we propose an automatic approach for the selection of relaxed and kept actions. Such an approach would be based on a data dependency analysis with heuristics for an optimized selection of relaxed actions. The development of adequate heuristics is still due to future research.

Appendix A

Excerpts from the Specification for CEPS

In this appendix, we give some excerpts from the μCRL specification of the Common Electronic Purse Specifications (CEPS).

Exemplary Abstraction of Summands

```
% Exemplary definition of sort CommandType. Bool and Nat are standard
func commandData:CommandCodeType#Nat#TxTypeType#Nat#Nat#Nat#Nat#Bool#
       Bool#Bool#CompletionCode#Bool#Nat#Nat#Nat->CommandType

% Exemplary declarations of operations on elements of sort CommandType.
% Rewrite rules are skipped.
map  getCommand:CommandType->CommandCodeType
     getCurrency:CommandType->Nat
     getTxType:CommandType->TxTypeType
     getDateTime:CommandType->Nat
     getAcqID:CommandType->Nat
     getLdvID:CommandType->Nat
     getLoadAmt:CommandType->Nat
     getReturnRcep:CommandType->Bool
     getUpdateSlot:CommandType->Bool
     getUpdateOthr:CommandType->Bool
     getCompCode:CommandType->CompletionCode
     getMacPresent:CommandType->Bool
     getMac_S2:CommandType->Nat
     getR_LSAM:CommandType->Nat
     getNewBalMax:CommandType->Nat

     updateCommand:CommandType#CommandCodeType->CommandType
     updateCurrency:CommandType#Nat->CommandType
     updateTxType:CommandType#TxTypeType->CommandType
     updateDateTime:CommandType#Nat->CommandType
     updateAcqID:CommandType#Nat->CommandType
     updateLdvID:CommandType#Nat->CommandType
     updateLoadAmt:CommandType#Nat->CommandType
     updateReturnRcep:CommandType#Bool->CommandType
```

```
      updateUpdateSlot:CommandType#Bool->CommandType
      updateUpdateOthr:CommandType#Bool->CommandType
      updateCompCode:CommandType#CompletionCode->CommandType
      updateMacPresent:CommandType#Bool->CommandType
      updateMac_S2:CommandType#Nat->CommandType
      updateR_LSAM:CommandType#Nat->CommandType
      updateNewBalMax:CommandType#Nat->CommandType
      eq:CommandType#CommandType->Bool

...
% Process and exemplary summand, not abstracted
proc X(s0:State,pSlotCount:Nat,pRefCurCount:Nat,pLogSize:Nat,pNT_Limit:Nat,
       vIssId:Nat,vCardId:Nat,vDateExp:Nat,vSlots:ArraySlotType16,
       slotsReported:ArraySlotsReportedType16,
       vRefCur:ArrayReferenceCurrencyType3,vTxLog_pLogSize:Nat,
       vTxLog_InUse:Nat,vTxLog_NextInsert:Nat,vTxLogEntry:LogArrayType,
       vNT:Nat,vDeactivated:Bool,vLocked:Bool,vLoadAmount:Nat,
       vSlotIndex:Nat,vCurrencySought:Nat,vSlotsAvailable:Nat,
       vSlotsReported:Nat,vLastAvailSlot:Nat,vLogInqActive:Bool,
       vLogIndex:Nat,vLastInqType:TxTypeType,
       vNewBalMax:Nat,reportSize:Nat,report:ReportArrayType,
       mPowerValue:PowerType,mAID:AidType,mInquiry:CommandType,
       mSlotInfo:ReplyType,mFCI:FCIType,mTxLogInfo:ReplyType,
       mInitLoadResp:ReplyType,mCredLoadResp:ReplyType,mIssID:Nat,
       mCardID:Nat,mDateExp:Nat,mRefCurIndex:Nat,mRefCurrCurrency:Nat,
       mRefCurBalMax:Nat) =

...
  sum(mInquiry2:CommandType,CepCommand(mInquiry2).
    X(x2p0(x2p1(x2p0(one))),pSlotCount,pRefCurCount,pLogSize,pNT_Limit,
      vIssId,vCardId,vDateExp,vSlots,slotsReported,vRefCur,vTxLog_pLogSize,
      vTxLog_InUse,vTxLog_NextInsert,vTxLogEntry,vNT,vDeactivated,vLocked,
      vLoadAmount,vSlotIndex,vCurrencySought,vSlotsAvailable,
      vSlotsReported,vLastAvailSlot,vLogInqActive,vLogIndex,vLastInqType,
      vNewBalMax,reportSize,report,mPowerValue,mAID,mInquiry2,mSlotInfo,
      mFCI,mTxLogInfo,mInitLoadResp,mCredLoadResp,mIssID,mCardID,mDateExp,
      mRefCurIndex,mRefCurrCurrency,mRefCurBalMax)
    <|and(eq(s0,x2p0(x2p1(x2p1(x2p0(one))))),
          and(and(and(eq(getCommand(mInquiry2),LOADINIT),
                      ge(vNT,pNT_Limit)),not(vDeactivated)),
              not(vLocked)))|>delta)+

% Sort for abstracted CommandType
sort CommandType_abstr
```

```
func TT_CommandType:->CommandType_abstr
     known:CommandType->CommandType_abstr

map  getCommand:CommandType_abstr->CommandCodeType_abstr
     getCurrency:CommandType_abstr->Nat_abstr
     getTxType:CommandType_abstr->TxTypeType_abstr
     getDateTime:CommandType_abstr->Nat_abstr
     getAcqID:CommandType_abstr->Nat_abstr
     getLdvID:CommandType_abstr->Nat_abstr
     getLoadAmt:CommandType_abstr->Nat_abstr
     getReturnRcep:CommandType_abstr->Bool_abstr
     getUpdateSlot:CommandType_abstr->Bool_abstr
     getUpdateOthr:CommandType_abstr->Bool_abstr
     getCompCode:CommandType_abstr->CompletionCode
     getMacPresent:CommandType_abstr->Bool_abstr
     getMac_S2:CommandType_abstr->Nat_abstr
     getR_LSAM:CommandType_abstr->Nat_abstr
     getNewBalMax:CommandType_abstr->Nat_abstr

     updateCommand:CommandType_abstr#CommandCodeType_abstr
       ->CommandType_abstr
     updateCurrency:CommandType_abstr#Nat_abstr->CommandType_abstr
     updateTxType:CommandType_abstr#TxTypeType_abstr->CommandType_abstr
     updateDateTime:CommandType_abstr#Nat_abstr->CommandType_abstr
     updateAcqID:CommandType_abstr#Nat_abstr->CommandType_abstr
     updateLdvID:CommandType_abstr#Nat_abstr->CommandType_abstr
     updateLoadAmt:CommandType_abstr#Nat_abstr->CommandType_abstr
     updateReturnRcep:CommandType_abstr#Bool_abstr->CommandType_abstr
     updateUpdateSlot:CommandType_abstr#Bool_abstr->CommandType_abstr
     updateUpdateOthr:CommandType_abstr#Bool_abstr->CommandType_abstr
     updateCompCode:CommandType_abstr#CompletionCode->CommandType_abstr
     updateMacPresent:CommandType_abstr#Bool_abstr->CommandType_abstr
     updateMac_S2:CommandType_abstr#Nat_abstr->CommandType_abstr
     updateR_LSAM:CommandType_abstr#Nat_abstr->CommandType_abstr
     updateNewBalMax:CommandType_abstr#Nat_abstr->CommandType_abstr
     eq:CommandType_abstr#CommandType_abstr->Bool_abstr

% Exemplary rewrite rule for abstracted equality operation
rew  eq(TT_CommandType,TT_CommandType) = TT_Bool
     eq(TT_CommandType,known(x))       = TT_Bool
     eq(known(x),TT_CommandType)       = TT_Bool
     eq(known(x),known(y))             = known(eq(x,y))
...
% Process and exemplary summand, not abstracted
```

```
proc X(s0:State_abstr,pSlotCount:Nat_abstr,pRefCurCount:Nat_abstr,
       pLogSize:Nat_abstr,pNT_Limit:Nat_abstr,vIssId:Nat_abstr,
       vCardId:Nat_abstr,vDateExp:Nat_abstr,vSlots:ArraySlotType16_abstr,
       slotsReported:ArraySlotsReportedType16_abstr,
       vRefCur:ArrayReferenceCurrencyType3_abstr,vTxLog_pLogSize:Nat_abstr,
       vTxLog_InUse:Nat_abstr,vTxLog_NextInsert:Nat_abstr,
       vTxLogEntry:LogArrayType_abstr,vNT:Nat_abstr,
       vDeactivated:Bool_abstr,vLocked:Bool_abstr,vLoadAmount:Nat_abstr,
       vSlotIndex:Nat_abstr,vCurrencySought:Nat_abstr,
       vSlotsAvailable:Nat_abstr,vSlotsReported:Nat_abstr,
       vLastAvailSlot:Nat_abstr,vLogInqActive:Bool_abstr,
       vLogIndex:Nat_abstr,vLastInqType:TxTypeType_abstr,
       vNewBalMax:Nat_abstr,reportSize:Nat_abstr,
       report:ReportArrayType_abstr,
       mPowerValue:PowerType_abstr,mAID:AidType_abstr,
       mInquiry:CommandType_abstr,mSlotInfo:ReplyType_abstr,
       mFCI:FCIType_abstr,mTxLogInfo:ReplyType_abstr,
       mInitLoadResp:ReplyType_abstr,mCredLoadResp:ReplyType_abstr,
       mIssID:Nat_abstr,mCardID:Nat_abstr,mDateExp:Nat_abstr,
       mRefCurIndex:Nat_abstr,mRefCurrCurrency:Nat_abstr,
       mRefCurBalMax:Nat) =
...

  CepCommand(TT_CommandType).
    X(x2p0(x2p1(x2p0(known(one)))),pSlotCount,pRefCurCount,pLogSize,
      pNT_Limit,vIssId,vCardId,vDateExp,vSlots,slotsReported,vRefCur,
      vTxLog_pLogSize,vTxLog_InUse,vTxLog_NextInsert,vTxLogEntry,vNT,
      vDeactivated,vLocked,vLoadAmount,vSlotIndex,vCurrencySought,
      vSlotsAvailable,vSlotsReported,vLastAvailSlot,vLogInqActive,
      vLogIndex,vLastInqType,vNewBalMax,reportSize,report,mPowerValue,
      mAID,TT_CommandType,mSlotInfo,mFCI,mTxLogInfo,mInitLoadResp,
      mCredLoadResp,mIssID,mCardID,mDateExp,mRefCurIndex,mRefCurrCurrency,
      mRefCurBalMax)
    <|may(and(eq(s0,x2p0(x2p1(x2p1(x2p0(known(one)))))),
              and(and(and(eq(getCommand(TT_CommandType),known(LOADINIT)),
                          ge(vNT,pNT_Limit)),not(vDeactivated)),
                  not(vLocked))))|>delta+
...
```

Exemplary CLP for CEPS

This section shows a part of the CLP for the specification of CEPS from the previous section. Due to the simplification described in Section 5.2.4, operations with a return type $\mathbb{B}$ are realized as rules with no return value but direct evaluation.

```
% Some operations on data of type CommandType
updateNewBalMax(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  nat(V15),commandData(commandCodeType(V0),nat(V1),txTypeType(V2),
  nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(VB),bool(V9),
  completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V15))).
updateR_LSAM(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  nat(V15),commandData(commandCodeType(V0),nat(V1),txTypeType(V2),
  nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),bool(V9),
  completionCode(V10),bool(V11),nat(V12),nat(V15),nat(V14))).
updateMac_S2(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  nat(V15),commandData(commandCodeType(V0),nat(V1),txTypeType(V2),
  nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),bool(V9),
  completionCode(V10),bool(V11),nat(V15),nat(V13),nat(V14))).
updateMacPresent(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  bool(V15),commandData(commandCodeType(V0),nat(V1),txTypeType(V2),
  nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),bool(V9),
  completionCode(V10),bool(V15),nat(V12),nat(V13),nat(V14))).
updateUpdateOthr(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  bool(V15),commandData(commandCodeType(V0),nat(V1),txTypeType(V2),
  nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),bool(V15),
  completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))).
updateUpdateSlot(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  bool(V15),commandData(commandCodeType(V0),nat(V1),txTypeType(V2),
  nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V15),bool(V9),
  completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))).
updateReturnRcep(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  bool(V15),commandData(commandCodeType(V0),nat(V1),txTypeType(V2),
  nat(V3),nat(V4),nat(V5),nat(V6),bool(V15),bool(V8),bool(V9),
  completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))).
updateLoadAmt(commandType(commandData(commandCodeType(V0),nat(V1),
```

```
    txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
    bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
    nat(V15),commandData(commandCodeType(V0),nat(V1),txTypeType(V2),
    nat(V3),nat(V4),nat(V5),nat(V15),bool(V7),bool(V8),bool(V9),
    completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))).
  updateLdvID(commandType(commandData(commandCodeType(V0),nat(V1),
    txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
    bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
    nat(V15),commandData(commandCodeType(V0),nat(V1),txTypeType(V2),
    nat(V3),nat(V4),nat(V15),nat(V6),bool(V7),bool(V8),bool(V9),
    completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))).
  updateAcqID(commandType(commandData(commandCodeType(V0),nat(V1),
    txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
    bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
    nat(V15),commandData(commandCodeType(V0),nat(V1),txTypeType(V2),
    nat(V3),nat(V15),nat(V5),nat(V6),bool(V7),bool(V8),bool(V9),
    completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))).
  updateDateTime(commandType(commandData(commandCodeType(V0),nat(V1),
    txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
    bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
    nat(V15),commandData(commandCodeType(V0),nat(V1),txTypeType(V2),
    nat(V15),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),bool(V9),
    completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))).
  updateTxType(commandType(commandData(commandCodeType(V0),nat(V1),
    txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
    bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
    txTypeType(V15),commandData(commandCodeType(V0),nat(V1),
    txTypeType(V15),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
    bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))).
  updateCommand(commandType(commandData(commandCodeType(V0),nat(V1),
    txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
    bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
    commandCodeType(V15),commandData(commandCodeType(V15),nat(V1),
    txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
    bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))).
  getNewBalMax(commandType(commandData(commandCodeType(V0),nat(V1),
    txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
    bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
    nat(V14)).
  getR_LSAM(commandType(commandData(commandCodeType(V0),nat(V1),
    txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
    bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
    nat(V13)).
  getMac_S2(commandType(commandData(commandCodeType(V0),nat(V1),
```

```
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  nat(V12)).
getMacPresent(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))))
  :-bool(V11).
getUpdateOthr(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))))
  :-bool(V9).
getUpdateSlot(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))))
  :-bool(V8).
getReturnRcep(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))))
  :-bool(V7).
getLoadAmt(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  nat(V6)).
getLdvID(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  nat(V5)).
getAcqID(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  nat(V4)).
getDateTime(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  nat(V3)).
getxType(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  txTypeType(V2)).
getCommand(commandType(commandData(commandCodeType(V0),nat(V1),
  txTypeType(V2),nat(V3),nat(V4),nat(V5),nat(V6),bool(V7),bool(V8),
  bool(V9),completionCode(V10),bool(V11),nat(V12),nat(V13),nat(V14))),
  commandType(V0)).
...
```

```
% The rule for the shown summand from Section A.A
cepCommand(global(state(Vs0),nat(VpSlotCount),nat(VpRefCurCount),
  nat(VpLogSize),nat(VpNT_Limit),nat(VvIssId),nat(VvCardId),nat(VvDateExp),
  arraySlotType16(VvSlots),arraySlotsReportedType16(VslotsReported),
  arrayReferenceCurrencyType3(VvRefCur),nat(VvTxLog_pLogSize),
  nat(VvTxLog_InUse),nat(VvTxLog_NextInsert),logArrayType(VvTxLogEntry),
  nat(VvNT),bool(VvDeactivated),bool(VvLocked),nat(VvLoadAmount),
  nat(VvSlotIndex),nat(VvCurrencySought),nat(VvSlotsAvailable),
  nat(VvSlotsReported),nat(VvLastAvailSlot),bool(VvLogInqActive),
  nat(VvLogIndex),txTypeType(VvLastInqType),nat(VvNewBalMax),
  nat(VreportSize),reportArrayType(Vreport),powerType(VmPowerValue),
  aidType(VmAID),commandType(VmInquiry),replyType(VmSlotInfo),
  fCIType(VmFCI),replyType(VmTxLogInfo),replyType(VmInitLoadResp),
  replyType(VmCredLoadResp),nat(VmIssID),nat(VmCardID),nat(VmDateExp),
  nat(VmRefCurIndex),nat(VmRefCurrCurrency),nat(VmRefCurBalMax)),
  global(state(x2p0(state(x2p1(state(x2p0(state(one)))))))),
  nat(VpSlotCount),nat(VpRefCurCount),nat(VpLogSize),nat(VpNT_Limit),
  nat(VvIssId),nat(VvCardId),nat(VvDateExp),arraySlotType16(VvSlots),
  arraySlotsReportedType16(VslotsReported),
  arrayReferenceCurrencyType3(VvRefCur),nat(VvTxLog_pLogSize),
  nat(VvTxLog_InUse),nat(VvTxLog_NextInsert),logArrayType(VvTxLogEntry),
  nat(VvNT),bool(VvDeactivated),bool(VvLocked),nat(VvLoadAmount),
  nat(VvSlotIndex),nat(VvCurrencySought),nat(VvSlotsAvailable),
  nat(VvSlotsReported),nat(VvLastAvailSlot),bool(VvLogInqActive),
  nat(VvLogIndex),txTypeType(VvLastInqType),nat(VvNewBalMax),
  nat(VreportSize),reportArrayType(Vreport),powerType(VmPowerValue),
  aidType(VmAID),commandType(VmInquiry2),replyType(VmSlotInfo),
  fCIType(VmFCI),replyType(VmTxLogInfo),replyType(VmInitLoadResp),
  replyType(VmCredLoadResp),nat(VmIssID),nat(VmCardID),nat(VmDateExp),
  nat(VmRefCurIndex),nat(VmRefCurrCurrency),nat(VmRefCurBalMax)),
  param(commandType(VmInquiry2))) :-

  and(bool(eq(state(Vs0),
       state(x2p0(state(x2p1(state(x2p1(state(x2p0(state(one))))))))))),
      bool(and(bool(and(bool(and(bool(eq(commandCodeType(
        getCommand(commandType(VmInquiry2))),commandCodeType(lOADINIT))),
      bool(ge(nat(VvNT),nat(VpNT_Limit))))),
      bool(not(bool(VvDeactivated))))),bool(not(bool(VvLocked))))).
```

Appendix B

Proofs for Lemma 3.22

Lemma B.1. Given a system of equations $\mathcal{E}$ and a corresponding CLP $\mathfrak{P}_{Adt}$, it holds that

$$s \Rightarrow_{TRS_i} t \Leftrightarrow transformTerm(s) \Rightarrow_{LP} transformTerm(t)$$

with $transformTerm(s) \Rightarrow_{LP} transformTerm(t)$ including substitution steps after the actual resolution of $transformTerm(s)$. ■

Proof. We have to prove that rewriting a term in a TRS can be simulated by the LP. Therefore, we have to prove, that for an arbitrary rewrite step in any of the two systems, the relation below holds:

$$\begin{array}{ccc} s & \longrightarrow & transformTerm(s) \\ {\scriptstyle TRS_i}\big\downarrow & & \big\downarrow \text{LP} \\ & & \Downarrow \text{SUB} \\ t & \longrightarrow & transformTerm(t) \end{array}$$

We will argument using a proof by induction over the depth of the redex in terms for constructors and functions (cf. Definition 3.21).

$s \Rightarrow_{TRS_i} t \Rightarrow transformTerm(s) \Rightarrow_{LP} transformTerm(t)$

Base Cases: The base cases are formed by those simulation cases, where a term s in the TRS can immediately be rewritten to another term t by an equation $s = t \in \mathcal{E}$.

Case 1. Let $f \in M$, $c \in C$, $f(s_1, \ldots, s_n) = c(t_1, \ldots, t_m) \in \mathcal{E}$ and $x, \hat{x}$ be fresh variables. Then there exists a rule $\rho \in \mathfrak{P}$ with

$$f(\mathbb{D}_1(s_1), \ldots, \mathbb{D}_n(s_n), \mathbb{D}(x)) \leftarrow x =_{\mathbb{D}} c(\mathbb{D}'_1(t_1), \ldots, \mathbb{D}'_m(t_m))$$

such that $t_j = s_i^\sigma$. Now assume, that $f(s_1, \ldots, s_n) \Rightarrow_{TRS_i} c(t_1, \ldots, t_m)$, which is already given by the fact that $f(s_1, \ldots, s_n) = c(t_1, \ldots, t_m) \in \mathcal{E}$. The actual rewriting takes place as $f(s_1, \ldots, s_n)$ appears as a term in the TRS. By Definition 3.15,

$$transformTerm(f(s_1, \ldots, s_n)) := f(\mathbb{D}_1(s_1), \ldots, \mathbb{D}_n(s_n), \mathbb{D}(x)).$$

The term, within which it is rewritten, results in a query to the CLP as follows:

$$\frac{\square \leftarrow f(\mathbb{D}_1(s_1),\ldots,\mathbb{D}_n(s_n),\mathbb{D}(\hat{x}))\wedge_p \ldots}{\square \leftarrow \hat{x} =_{\mathbb{D}} c(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m))\wedge_p \ldots}\rightarrow_{LP}$$

The Prolog representation of $f(s_1,\ldots,s_n)$ is resolved to

$$\hat{x} =_{\mathbb{D}} c(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m)),$$

which is $transformTerm(c(t_1,\ldots,t_m))$. For this, we have

$$transformTerm(f(s_1,\ldots,s_n)) \Rightarrow_{LP} transformTerm(c(t_1,\ldots,t_m)).$$

Case 2. Let $f,g \in \mathcal{M}$, $f(s_1,\ldots,s_n) = g(t_1,\ldots,t_m) \in \mathcal{E}$ and $x,\hat{x}$ be fresh variables. Then there exists a rule $\rho \in \mathfrak{P}$ with

$$f(\mathbb{D}_1(s_1),\ldots,\mathbb{D}_n(s_n),\mathbb{D}(x_1)) \leftarrow x_1 =_{\mathbb{D}} x_2 \wedge_p g(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m),\mathbb{D}(x_2))$$

such that $t_j = s_i^\sigma$. Now assume, that $f(s_1,\ldots,s_n) \Rightarrow_{TRS_i} g(t_1,\ldots,t_m)$, which is already given by the fact that $f(s_1,\ldots,s_n) = g(t_1,\ldots,t_m) \in \mathcal{E}$. The actual rewriting takes place as $f(s_1,\ldots,s_n)$ appears as a term in the TRS. By Definition 3.15,

$$transformTerm(f(s_1,\ldots,s_n)) := f(\mathbb{D}_1(s_1),\ldots,\mathbb{D}_n(s_n),\mathbb{D}(x)).$$

The term, within which it is rewritten, results in a query to the CLP as follows:

$$\cfrac{\cfrac{\square \leftarrow f(\mathbb{D}_1(s_1),\ldots,\mathbb{D}_n(s_n),\mathbb{D}(\hat{x}))\wedge_p \ldots}{\square \leftarrow \hat{x} =_{\mathbb{D}} \hat{x}'\wedge_p\ g(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m),\mathbb{D}(\hat{x}'))\wedge_p \ldots}\rightarrow_{LP}}{\square \leftarrow \hat{x} =_{\mathbb{D}} \hat{x}\wedge_p\ g(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m),\mathbb{D}(\hat{x}))\wedge_p \ldots}\rightarrow_{SUB[\hat{x}'/\hat{x}]}$$
$$\square \leftarrow g(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m),\mathbb{D}(\hat{x}))\wedge_p \ldots$$

The Prolog representation of $f(s_1,\ldots,s_n)$ is resolved to

$$g(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m),\mathbb{D}(\hat{x})),$$

which is $transformTerm(g(t_1,\ldots,t_m))$. For this, we have

$$transformTerm(f(s_1,\ldots,s_n)) \Rightarrow_{LP} transformTerm(g(t_1,\ldots,t_m)).$$

Inductive Cases: Now assume that

$$s \Rightarrow_{TRS_i} t \Rightarrow transformTerm(s) \Rightarrow_{LP} transformTerm(t).$$

Let $f(\ldots,s,\ldots) = f(\ldots,t,\ldots) \in \mathcal{E}$, such that $s \Rightarrow_{TRS_i} t$ while $f(\ldots,s,\ldots) \Rightarrow_{TRS_i} f(\ldots,t,\ldots)$.

Case 3. Let $s := g(s_1, \ldots, s_n)$ and $g \in M$, $t := h(t_1, \ldots, t_m)$ and $h \in M$ and $x, y, \hat{x}, \hat{y}$ be fresh variables. By Definition 3.15,

$$\begin{aligned} &\mathit{transformTerm}(f(\ldots, g(s_1, \ldots, s_n), \ldots)) := \\ &\qquad \ldots \wedge_p g(\mathbb{D}_1(\hat{s}_1), \ldots, \mathbb{D}_n(\hat{s}_n), \mathbb{D}_s(x)) \wedge_p \\ &\qquad \ldots \wedge_p f(\ldots, \mathbb{D}_s(x), \ldots, \mathbb{D}(y)) \end{aligned}$$

with $\mathit{transformTerm}(g(s_1, \ldots, s_n)) := g(\mathbb{D}_1(\hat{s}_1), \ldots, \mathbb{D}_n(\hat{s}_n), \mathbb{D}_s(x))$ and $\hat{s}_i = \mathit{transformTerm}(s_i)$.

By construction, there exists a rule $\rho \in \mathfrak{P}$ with

$$\begin{aligned} g(\mathbb{D}_1(\hat{s}_1), \ldots, \mathbb{D}_n(\hat{s}_n), \mathbb{D}_s(x_1)) \leftarrow & x_1 =_{\mathbb{D}} x_2 \wedge_p \\ & h(\mathbb{D}'_1(\hat{t}_1), \ldots, \mathbb{D}'_m(\hat{t}_m), \mathbb{D}_s(x_2)) \end{aligned}$$

such that $\hat{t}_j = \hat{s}_i^{\sigma}$. The simulation of term rewriting takes place in a query. This query has, by induction, the following resolution steps (we skip the leading dots here, since resolution takes place for the first resolvent only):

$$\begin{array}{l} \square \leftarrow g(\mathbb{D}_1(\hat{s}_1), \ldots, \mathbb{D}_n(\hat{s}_n), \mathbb{D}_s(\hat{x})) \wedge_p \ldots \wedge_p \\ \qquad f(\ldots, \mathbb{D}_s(\hat{x}), \ldots, \mathbb{D}(\hat{y})) \wedge_p \ldots \\ \hline \square \leftarrow \hat{x} =_{\mathbb{D}} \hat{x}' \wedge_p \\ \qquad h(\mathbb{D}'_1(\hat{t}_1), \ldots, \mathbb{D}'_m(\hat{t}_m), \mathbb{D}_s(\hat{x}')) \wedge_p \ldots \wedge_p \\ \qquad f(\ldots, \mathbb{D}_s(\hat{x}), \ldots, \mathbb{D}(\hat{y})) \wedge_p \ldots \\ \hline \square \leftarrow \hat{x} =_{\mathbb{D}} \hat{x} \wedge_p \\ \qquad h(\mathbb{D}'_1(\hat{t}_1), \ldots, \mathbb{D}'_m(\hat{t}_m), \mathbb{D}_s(\hat{x})) \wedge_p \ldots \wedge_p \\ \qquad f(\ldots, \mathbb{D}_s(\hat{x}), \ldots, \mathbb{D}(\hat{y})) \wedge_p \ldots \\ \hline \square \leftarrow h(\mathbb{D}'_1(\hat{t}_1), \ldots, \mathbb{D}'_m(\hat{t}_m), \mathbb{D}_s(\hat{x})) \wedge_p \ldots \wedge_p \\ \qquad f(\ldots, \mathbb{D}_s(\hat{x}), \ldots, \mathbb{D}(\hat{y})) \wedge_p \ldots \end{array} \begin{array}{l} \\ \rightarrow_{LP} \\ \\ \\ \rightarrow_{SUB[\hat{x}'/\hat{x}]} \\ \\ \\ \\ \end{array}$$

The Prolog representation of $g(s_1, \ldots, s_n)$ is resolved to

$$h(\mathbb{D}'_1(\hat{t}_1), \ldots, \mathbb{D}'_m(\hat{t}_m), \mathbb{D}_s(\hat{x})),$$

which is $\mathit{transformTerm}(h(t_1, \ldots, t_m))$. The Prolog representation of

$$f(\ldots, g(s_1, \ldots, s_n), \ldots)$$

is by that resolved to

$$h(\mathbb{D}'_1(\hat{t}_1), \ldots, \mathbb{D}'_m(\hat{t}_m), \mathbb{D}_s(\hat{x})) \wedge_p \ldots \wedge_p f(\ldots, \mathbb{D}_s(\hat{x}), \ldots, \mathbb{D}(\hat{y})),$$

which is $\mathit{transformTerm}(f(\ldots, h(t_1, \ldots, t_m), \ldots))$. For this, we have

$$\mathit{transformTerm}(s) \Rightarrow_{LP} \mathit{transformTerm}(t)$$

while $\mathit{transformTerm}(f(\ldots, s, \ldots)) \Rightarrow_{LP} \mathit{transformTerm}(g(\ldots, t, \ldots))$.

Case 4. Let $s := g(s_1, \ldots, s_n)$ and $g \in M$, $t := c(t_1, \ldots, t_m)$ and $c \in C$ and $x, y, \hat{x}, \hat{y}$ be fresh variables. By Definition 3.15,

$$\begin{aligned} &transformTerm(f(\ldots, g(s_1, \ldots, s_n), \ldots)) := \\ &\qquad \ldots \wedge_p g(\mathbb{D}_1(\hat{s}_1), \ldots, \mathbb{D}_n(\hat{s}_n), \mathbb{D}_s(x)) \wedge_p \\ &\qquad\qquad \ldots \wedge_p f(\ldots, \mathbb{D}_s(x), \ldots, \mathbb{D}(y)) \end{aligned}$$

with

$$transformTerm(g(s_1, \ldots, s_n)) := g(\mathbb{D}_1(\hat{s}_1), \ldots, \mathbb{D}_n(\hat{s}_n), \mathbb{D}_s(x))$$

and $\hat{s}_i = transformTerm(s_i)$.

By construction, there exists a rule $\rho \in \mathfrak{P}$ with

$$g(\mathbb{D}_1(\hat{s}_1), \ldots, \mathbb{D}_n(\hat{s}_n), \mathbb{D}_s(x)) \leftarrow x =_{\mathbb{D}_s} h(\mathbb{D}'_1(\hat{t}_1), \ldots, \mathbb{D}'_m(\hat{t}_m))$$

such that $\hat{t}_j = \hat{s}_i^{\sigma}$. The simulation of term rewriting takes place in a query. This query has, by induction, the following resolution steps (we skip the leading dots here, since resolution takes place for the first resolvent only):

$$\frac{\begin{array}{l} \square \leftarrow g(\mathbb{D}_1(\hat{s}_1), \ldots, \mathbb{D}_n(\hat{s}_n), \mathbb{D}_s(\hat{x})) \wedge_p \ldots \wedge_p \\ \qquad f(\ldots, \mathbb{D}_s(\hat{x}), \ldots, \mathbb{D}(\hat{y})) \wedge_p \ldots \end{array}}{\begin{array}{l} \square \leftarrow \hat{x} =_{\mathbb{D}_s} h(\mathbb{D}'_1(\hat{t}_1), \ldots, \mathbb{D}'_m(\hat{t}_m)) \wedge_p \ldots \wedge_p \\ \qquad f(\ldots, \mathbb{D}_s(\hat{x}), \ldots, \mathbb{D}(\hat{y})) \wedge_p \ldots \end{array}} \rightarrow_{LP}$$

The Prolog representation of $g(s_1, \ldots, s_n)$ is resolved to

$$\hat{x} =_{\mathbb{D}_s} h(\mathbb{D}'_1(\hat{t}_1), \ldots, \mathbb{D}'_m(\hat{t}_m)),$$

which is $transformTerm(h(t_1, \ldots, t_m))$. The Prolog representation of

$$f(\ldots, g(s_1, \ldots, s_n), \ldots)$$

is by that resolved to

$$\hat{x} =_{\mathbb{D}_s} h(\mathbb{D}'_1(\hat{t}_1), \ldots, \mathbb{D}'_m(\hat{t}_m)) \wedge_p \ldots \wedge_p f(\ldots, \mathbb{D}_s(\hat{x}), \ldots, \mathbb{D}(\hat{y})),$$

which is $transformTerm(f(\ldots, h(t_1, \ldots, t_m), \ldots))$. For this, we have

$$transformTerm(s) \Rightarrow_{LP} transformTerm(t)$$

while $transformTerm(f(\ldots, s, \ldots)) \Rightarrow_{LP} transformTerm(g(\ldots, t, \ldots))$.

Case 5. Let $s := c(s_1, \ldots, s_i, \ldots, s_n)$ and $c \in C$, $s_i := f(t_1, \ldots, t_m)$ and $f \in M$ and $x, y, \hat{x}, \hat{y}$ be fresh variables.

By Definition 3.15,

$$\begin{aligned} \mathit{transformTerm}(c(s_1,\ldots,f(t_1,\ldots,t_m),\ldots,s_n)) := \\ \ldots \wedge_p f(\mathbb{D}_{t_1}(\hat{t}_1),\ldots,\mathbb{D}_{t_m}(\hat{t}_m),\mathbb{D}_f(y)) \wedge_p \\ \ldots \wedge_p x =_{\mathbb{D}} c(\mathbb{D}_{s_1}(\hat{s}_1),\ldots,\mathbb{D}_f(y),\ldots,\mathbb{D}_{s_n}(\hat{s}_n)). \end{aligned}$$

By that, the simulation of term rewriting takes place in the following query:

$$\dfrac{\dfrac{\dfrac{\square \leftarrow \ldots \wedge_p f(\mathbb{D}_{t_1}(\hat{t}_1),\ldots,\mathbb{D}_{t_m}(\hat{t}_m),\mathbb{D}_f(\hat{y})) \wedge_p \ldots \wedge_p \hat{x} =_{\mathbb{D}} c(\mathbb{D}_{s_1}(\hat{s}_1),\ldots,\mathbb{D}_f(\hat{y}),\ldots,\mathbb{D}_{s_n}(\hat{s}_n))}{\square \leftarrow f(\mathbb{D}_{t_1}(\hat{t}_1),\ldots,\mathbb{D}_{t_m}(\hat{t}_m),\mathbb{D}_f(\hat{y})) \wedge_p \ldots \wedge_p \hat{x} =_{\mathbb{D}} c(\mathbb{D}_{s_1}(\hat{s}_1),\ldots,\mathbb{D}_f(\hat{y}),\ldots,\mathbb{D}_{s_n}(\hat{s}_n))}\ \text{inductively}}{\square \leftarrow \ldots \wedge_p \hat{x} =_{\mathbb{D}} c(\mathbb{D}_{s_1}(\hat{s}_1),\ldots,\mathbb{D}_f(\hat{y}),\ldots,\mathbb{D}_{s_n}(\hat{s}_n))}\ \rightarrow_{LP}}$$

with $\hat{y}$ instantiated

By Definition 3.19, there exists a rule in $\mathfrak{P}_{Adt}$, such that rewriting $f(t_1,\ldots,t_m)$ can be simulated according to the other cases of this proof. In doing so, s is rewritten inductively.

Case 6. This case is an inductive one, too, with $s := c_1(s_1,\ldots,s_i,\ldots,s_n)$ and $s_i := c_2(\ldots,f(t_1,\ldots,t_m),\ldots)$, $c_1, c_2 \in C$ and $f \in M$. The term s is rewritten as in the previous case, leaving $c_2(\ldots)$ in place and tearing apart function f as has been shown in Case 5.

$s \Rightarrow_{TRS_i} t \Leftarrow \mathit{transformTerm}(s) \Rightarrow_{LP} \mathit{transformTerm}(t)$

Base Cases:

Case 1. Let $f \in M$, $c \in C$ and $x, \hat{x}$ be fresh variables. Assume further, we have a resolution step as follows:

$$\dfrac{\square \leftarrow f(\mathbb{D}_1(s_1),\ldots,\mathbb{D}_n(s_n),\mathbb{D}(\hat{x})) \wedge_p \ldots}{\square \leftarrow \hat{x} =_{\mathbb{D}} c(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m)) \wedge_p \ldots}\ \rightarrow_{LP}$$

The Prolog representation of $f(s_1,\ldots,s_n)$ is resolved to

$$\hat{x} =_{\mathbb{D}} c(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m)),$$

which is $\mathit{transformTerm}(c(t_1,\ldots,t_m))$. For this, we have

$$\mathit{transformTerm}(f(s_1,\ldots,s_n)) \Rightarrow_{LP} \mathit{transformTerm}(c(t_1,\ldots,t_m)).$$

This step is only possible, if there exists a rule $\rho \in \mathfrak{P}$ with

$$f(\mathbb{D}_1(s_1),\ldots,\mathbb{D}_n(s_n),\mathbb{D}(x)) \leftarrow x =_{\mathbb{D}} c(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m))$$

such that $t_j = s_i^\sigma$. By Definition 3.19, this rule exists in the CLP, if we have $f(s_1,\ldots,s_n) = c(t_1,\ldots,t_m) \in \mathcal{E}$. Hence, we can derive that also

$$f(s_1,\ldots,s_n) \Rightarrow_{TRS_i} c(t_1,\ldots,t_m).$$

Case 2. Let $f, g \in M$ and $x, \hat{x}$ be fresh variables. Assume further, we have a resolution step as follows:

$$\frac{\begin{array}{l}\square \leftarrow f(\mathbb{D}_1(s_1),\ldots,\mathbb{D}_n(s_n),\mathbb{D}(\hat{x})) \wedge_p \ldots\end{array}}{\begin{array}{l}\square \leftarrow \hat{x} =_{\mathbb{D}} \hat{x}' \wedge_p \\ \quad g(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m),\mathbb{D}(\hat{x}')) \wedge_p \ldots\end{array}}\ {\to_{LP}}$$

$$\frac{\begin{array}{l}\square \leftarrow \hat{x} =_{\mathbb{D}} \hat{x}' \wedge_p \\ \quad g(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m),\mathbb{D}(\hat{x}')) \wedge_p \ldots\end{array}}{\begin{array}{l}\square \leftarrow \hat{x} =_{\mathbb{D}} \hat{x} \wedge_p \\ \quad g(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m),\mathbb{D}(\hat{x})) \wedge_p \ldots\end{array}}\ {\to_{SUB[\hat{x}'/\hat{x}]}}$$

$$\begin{array}{l}\square \leftarrow g(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m),\mathbb{D}(\hat{x})) \wedge_p \ldots\end{array}$$

The Prolog representation of $f(s_1,\ldots,s_n)$ is resolved to

$$g(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m),\mathbb{D}(\hat{x})),$$

which is $transformTerm(g(t_1,\ldots,t_m))$. For this, we have

$$transformTerm(f(s_1,\ldots,s_n)) \Rightarrow_{LP} transformTerm(c(t_1,\ldots,t_m)).$$

This step is only possible, if there exists a rule $\rho \in \mathfrak{P}$ with

$$f(\mathbb{D}_1(s_1),\ldots,\mathbb{D}_n(s_n),\mathbb{D}(x_1)) \leftarrow x_1 =_{\mathbb{D}} x_2 \wedge_p g(\mathbb{D}'_1(t_1),\ldots,\mathbb{D}'_m(t_m),\mathbb{D}(x_2))$$

such that $t_j = s_i^\sigma$. By Definition 3.19, this rule exists in the CLP, if we have $f(s_1,\ldots,s_n) = g(t_1,\ldots,t_m) \in \mathcal{E}$. Hence, we can derive that also

$$f(s_1,\ldots,s_n) \Rightarrow_{TRS_i} g(t_1,\ldots,t_m).$$

Inductive Cases:

Case 3. Let $f, g, h \in M$ and $x, y, \hat{x}, \hat{y}$ be fresh variables. Assume further, we have a resolution step as follows:

$$\frac{\begin{array}{l}\square \leftarrow g(\mathbb{D}_1(\hat{s}_1),\ldots,\mathbb{D}_n(\hat{s}_n),\mathbb{D}_s(\hat{x})) \wedge_p \ldots \wedge_p \\ \quad f(\ldots,\mathbb{D}_s(\hat{x}),\ldots,\mathbb{D}(\hat{y})) \wedge_p \ldots\end{array}}{\begin{array}{l}\square \leftarrow \hat{x} =_{\mathbb{D}} \hat{x}' \wedge_p \\ \quad h(\mathbb{D}'_1(\hat{t}_1),\ldots,\mathbb{D}'_m(\hat{t}_m),\mathbb{D}_s(\hat{x}')) \wedge_p \ldots \wedge_p \\ \quad f(\ldots,\mathbb{D}_s(\hat{x}),\ldots,\mathbb{D}(\hat{y})) \wedge_p \ldots\end{array}}\ {\to_{LP}}$$

$$\frac{\begin{array}{l}\square \leftarrow \hat{x} =_{\mathbb{D}} \hat{x}' \wedge_p \\ \quad h(\mathbb{D}'_1(\hat{t}_1),\ldots,\mathbb{D}'_m(\hat{t}_m),\mathbb{D}_s(\hat{x}')) \wedge_p \ldots \wedge_p \\ \quad f(\ldots,\mathbb{D}_s(\hat{x}),\ldots,\mathbb{D}(\hat{y})) \wedge_p \ldots\end{array}}{\begin{array}{l}\square \leftarrow \hat{x} =_{\mathbb{D}} \hat{x} \wedge_p \\ \quad h(\mathbb{D}'_1(\hat{t}_1),\ldots,\mathbb{D}'_m(\hat{t}_m),\mathbb{D}_s(\hat{x})) \wedge_p \ldots \wedge_p \\ \quad f(\ldots,\mathbb{D}_s(\hat{x}),\ldots,\mathbb{D}(\hat{y})) \wedge_p \ldots\end{array}}\ {\to_{SUB[\hat{x}'/\hat{x}]}}$$

$$\begin{array}{l}\square \leftarrow h(\mathbb{D}'_1(\hat{t}_1),\ldots,\mathbb{D}'_m(\hat{t}_m),\mathbb{D}_s(\hat{x})) \wedge_p \ldots \wedge_p \\ \quad f(\ldots,\mathbb{D}_s(\hat{x}),\ldots,\mathbb{D}(\hat{y})) \wedge_p \ldots\end{array}$$

The Prolog representation of $g(s_1,\ldots,s_n)$ is resolved to

$$h(\mathbb{D}'_1(\hat{t}_1),\ldots,\mathbb{D}'_m(\hat{t}_m),\mathbb{D}_s(\hat{x})),$$

which is $transformTerm(h(t_1,\ldots,t_m))$. The Prolog representation of

$$f(\ldots,g(s_1,\ldots,s_n),\ldots)$$

is by that resolved to

$$h(\mathbb{D}'_1(\hat{t}_1),\ldots,\mathbb{D}'_m(\hat{t}_m),\mathbb{D}_s(\hat{x}))\wedge_p\ldots\wedge_p f(\ldots,\mathbb{D}_s(\hat{x}),\ldots,\mathbb{D}(\hat{y})),$$

which is $transformTerm(f(\ldots,h(t_1,\ldots,t_m),\ldots))$. For this, we have

$$transformTerm(s)\Rightarrow_{LP} transformTerm(t)$$

while $transformTerm(f(\ldots,s,\ldots))\Rightarrow_{LP} transformTerm(g(\ldots,t,\ldots))$. This step is only possible, if there exists a rule $\rho\in\mathfrak{P}$ with

$$g(\mathbb{D}_1(\hat{s}_1),\ldots,\mathbb{D}_n(\hat{s}_n),\mathbb{D}_s(x_1))\leftarrow x_1=_{\mathbb{D}} x_2\wedge_p h(\mathbb{D}'_1(\hat{t}_1),\ldots,\mathbb{D}'_m(\hat{t}_m),\mathbb{D}_s(x_2))$$

such that $\hat{t}_i=\hat{s}_i^{\sigma}$. By Definition 3.19, this rule exists in the CLP, if we have $g(s_1,\ldots,s_n)=h(t_1,\ldots,t_m)\in\mathcal{E}$. Hence, we can derive that also

$$g(s_1,\ldots,s_n)\Rightarrow_{TRS_i} h(t_1,\ldots,t_m)$$

while $f(\ldots,g(s_1,\ldots,s_n),\ldots)\Rightarrow_{TRS_i} f(\ldots,h(t_1,\ldots,t_m),\ldots)$.

Case 4. Let $f,g,h\in M$ and $x,y,\hat{x},\hat{y}$ be fresh variables. Assume further, we have a resolution step as follows:

$$\frac{\begin{array}{l}\square\leftarrow g(\mathbb{D}_1(\hat{s}_1),\ldots,\mathbb{D}_n(\hat{s}_n),\mathbb{D}_s(\hat{x}))\wedge_p\ldots\wedge_p\\ \qquad f(\ldots,\mathbb{D}_s(\hat{x}),\ldots,\mathbb{D}(\hat{y}))\wedge_p\ldots\end{array}}{\begin{array}{l}\square\leftarrow \hat{x}=_{\mathbb{D}_s} h(\mathbb{D}'_1(\hat{t}_1),\ldots,\mathbb{D}'_m(\hat{t}_m))\wedge_p\ldots\wedge_p\\ \qquad f(\ldots,\mathbb{D}_s(\hat{x}),\ldots,\mathbb{D}(\hat{y}))\wedge_p\ldots\end{array}}\rightarrow_{LP}$$

The Prolog representation of $g(s_1,\ldots,s_n)$ is resolved to

$$\hat{x}=_{\mathbb{D}_s} h(\mathbb{D}'_1(\hat{t}_1),\ldots,\mathbb{D}'_m(\hat{t}_m)),$$

which is $transformTerm(h(t_1,\ldots,t_m))$. The Prolog representation of

$$f(\ldots,g(s_1,\ldots,s_n),\ldots)$$

is by that resolved to

$$\hat{x}=_{\mathbb{D}_s} h(\mathbb{D}'_1(\hat{t}_1),\ldots,\mathbb{D}'_m(\hat{t}_m))\wedge_p\ldots\wedge_p f(\ldots,\mathbb{D}_s(\hat{x}),\ldots,\mathbb{D}(\hat{y})),$$

which is $\mathit{transformTerm}(f(\ldots, h(t_1, \ldots, t_m), \ldots))$. For this, we have

$$\mathit{transformTerm}(s) \Rightarrow_{LP} \mathit{transformTerm}(t)$$

while $\mathit{transformTerm}(f(\ldots, s, \ldots)) \Rightarrow_{LP} \mathit{transformTerm}(g(\ldots, t, \ldots))$. This step is only possible, if there exists a rule $\rho \in \mathfrak{P}$ with

$$g(\mathbb{D}_1(\hat{s}_1), \ldots, \mathbb{D}_n(\hat{s}_n), \mathbb{D}(x)) \leftarrow x =_{\mathbb{D}_s} h(\mathbb{D}'_1(\hat{t}_1), \ldots, \mathbb{D}'_m(\hat{t}_m), \mathbb{D}(x))$$

such that $\hat{t}_i = \hat{s}_i^{\sigma}$. By Definition 3.19, this rule exists in the CLP, if we have $g(s_1, \ldots, s_n) = h(t_1, \ldots, t_m) \in \mathcal{E}$. Hence, we can derive that also

$$g(s_1, \ldots, s_n) \Rightarrow_{TRS_i} h(t_1, \ldots, t_m)$$

while $f(\ldots, g(s_1, \ldots, s_n), \ldots) \Rightarrow_{TRS_i} f(\ldots, h(t_1, \ldots, t_m), \ldots)$.

Case 5. Let $c \in C$, $f \in M$ and $x, y, \hat{x}, \hat{y}$ be fresh variables. Assume further, we have a resolution step as follows:

$$\dfrac{\dfrac{\square \leftarrow \ldots \wedge_p f(\mathbb{D}_{t_1}(\hat{t}_1), \ldots, \mathbb{D}_{t_m}(\hat{t}_m), \mathbb{D}_f(\hat{y})) \wedge_p \ldots \wedge_p \hat{x} =_{\mathbb{D}} c(\mathbb{D}_{s_1}(\hat{s}_1), \ldots, \mathbb{D}_f(\hat{y}), \ldots, \mathbb{D}_{s_n}(\hat{s}_n))}{\square \leftarrow f(\mathbb{D}_{t_1}(\hat{t}_1), \ldots, \mathbb{D}_{t_m}(\hat{t}_m), \mathbb{D}_f(\hat{y})) \wedge_p \ldots \wedge_p \hat{x} =_{\mathbb{D}} c(\mathbb{D}_{s_1}(\hat{s}_1), \ldots, \mathbb{D}_f(\hat{y}), \ldots, \mathbb{D}_{s_n}(\hat{s}_n))}\ \text{inductively}}{\square \leftarrow \ldots \wedge_p \hat{x} =_{\mathbb{D}} c(\mathbb{D}_{s_1}(\hat{s}_1), \ldots, \mathbb{D}_f(\hat{y}), \ldots, \mathbb{D}_{s_n}(\hat{s}_n)) \text{ with } \hat{y} \text{ instantiated}}\ \rightarrow_{LP}$$

This step is only possible, if there exists a rule in $\mathfrak{P}_{Adt}$, such that rewriting $f(t_1, \ldots, t_m)$ can be simulated according to the other cases of this proof. This rule exists by Definition 3.19. Hence, we can derive that also $c(\ldots, s_i, \ldots) \Rightarrow_{TRS_i} c(\ldots, s'_i, \ldots)$ while $s_i \Rightarrow_{TRS_i} s'_i$ and $s_i := f(t_1, \ldots, t_m)$.

□

As a second proof, we have to show, that also multi-step term rewritings are treated correctly.

Proof. We will argument using a proof by induction over the number of single rewriting steps necessary to do the complete rewriting of a term.

Base Case: Assume, we have a rewrite step $s \rightarrow_{TRS_i} s'$ with $s := f(t_1, \ldots, t_m)$, $f \in M$, and $s' := c(t'_1, \ldots, t'_m)$, $c \in C$. Then, rewriting takes place as shown in the previous proof, Case 1.

Inductive Case: Assume, we have a number of rewriting steps $s \rightarrow_{TRS_i} s' \rightarrow_{TRS_i} \ldots \rightarrow_{TRS_i} s'' \rightarrow_{TRS_i} s'''$. Then, a step-wise rewriting process subsequently takes place to rewrite $s \Rightarrow_{TRS_i} s''$ according to the Cases 2–6 of the previous proof. The last rewrite step $s'' \rightarrow_{TRS_i} s'''$ is the one from the base case of this proof.

□

Acronyms

ADT	Abstract Datatype
AJAX	Asynchronous JavaScript and XML
ALTL	Action-based Linear Temporal Logic
API	Application Programming Interface
ATC	Abstract Test Case
ATM	Automatic Teller Machine
BAiT	Behavior-Adaptation in Testing
BRICKS	Basic Research in Informatics for Creating the Knowledge Society
CADP	Construction and Analysis of Distributed Processes
CASE	Computer-Aided Software Engineering
CEPS	Common Electronic Purse Specifications
CHR	Constraint Handling Rules
CLP	Constraint Logic Program
CSS	Cascading Style Sheet
CTG	Complete Test Graph
DOM	Document Object Model
eALTL	Extended Action-based Linear Temporal Logic
EFSM	Extended Finite State Machine
FSM	Finite State Machine
HTML	Hypertext Markup Language
IOLTS	Input/Output Labeled Transition System
IOSTS	Input/Output Symbolic Transition System
ITEA	Information Technology for European Advancement
IUT	Implementation under Test
LPE	Linear Process Equation
LP	Logic Program
LTL	Linear Temporal Logic
LTS	Labeled Transition System
μCRL	micro Common Representation Language
MSC	Message Sequence Chart
OMG	Object Management Group

PAR	Positive Acknowledgement Retransmission Protocol
PIN	Personal Identification Number
SDL	Specification and Description Language
STG	Symbolic Test Generation
STS	Symbolic Transition System
SUT	System under Test
SWT	Standard Widget Toolkit
TAIO	Timed Automaton with Inputs and Outputs
TCI-CD	TTCN-3 Control Interface, Coding/Decoding Interface
TGV	Test Generation with Verification Techniques
TIOLTS	Timed Input/Output Labeled Transition System
TRI	TTCN-3 Runtime Interface
TRS	Term Rewriting System
TTCN-3	Testing and Test Control Notation, version 3
TT-Medal	Tests & Testing Methodologies for Advanced Languages
UML	Unified Modeling Language
W3C	World Wide Web Consortium
XPCOM	Cross Platform Component Object Model
xUnit	Unit Testing Frameworks

Symbols

Miscellaneous

$\mathbb{B}$	Boolean; $\mathbb{B} = \{\top, \bot\}$.	
$\mathbb{N}$	Natural numbers; $\mathbb{N} = \{n \mid n \geqslant 0\}$.	
$\mathbb{Z}$	Integers *(positive and negative numbers)*.	
$\mathbb{D}$	Arbitrary datatype/variable domain.	
$\mathcal{D}$	Set of arbitrary datatype/variable domains.	
2^S	Power set of an arbitrary set S.	
S^*	Words from an arbitrary set S.	
ρ, ξ, ν	Rules in a CLP.	
s, t	Terms.	
v, x, y, z	Variables.	

Specifications and Automata

$\mathfrak{S}$	A specification; here: a Symbolic Transition System.	pp. 14; 25
$\mathtt{Var}$	Set of variables in the STS.	p. 14
$\mathtt{L}$	Set of locations in the STS.	p. 14
$\mathtt{Val}$	Set of valuations in the STS.	p. 14
$\mathtt{A}$	Set of actions in the STS.	p. 14
$\mathtt{E}$	Set of edges in the STS.	p. 14
ℓ, ℓ_{init}	(Initial) location in the STS.	p. 14
η, η_{init}	(Initial) valuation in a state of the STS.	p. 14
$\xrightarrow{\iota}$	Single edge in an STS, labeled with action ι.	p. 15
$\xrightarrow{g \triangleright x := e}$	Assignment of e to x under condition (guard) g.	p. 15
$\xrightarrow{g \triangleright !s(e)}$	Output of event s with parameter e under condition g.	p. 15
$\xrightarrow{g \triangleright ?s(x)}$	Input of event s with parameter x under condition g.	p. 15
$\mathfrak{M}$	A Labeled Transition System.	p. 16
Σ	Set of states in the LTS.	p. 16
Λ	Set of labels in the LTS.	p. 16
Δ	Set of transitions in the LTS.	p. 16
$\xrightarrow{\lambda}$	Single transition labeled with $\lambda \in \Lambda$.	p. 16
σ, σ_{init}	(Initial) state in the LTS.	p. 16
τ	Internal action (τ-step).	p. 16
δ	Deadlock.	p. 58
π	Trace.	p. 17
$\lvert\pi\rvert$	Length of a finite trace π.	p. 17
$[\![\mathfrak{M}]\!]_{\text{traces}}$	Set of traces of $\mathfrak{M}$.	p. 17

$\mathbb{S}$	A sort in a μCRL specification.	pp. 25; 22
E	Set of equations in a μCRL specification.	p. 25
S	Set of sorts in a μCRL specification.	pp. 25; 26
$\mathcal{F} = C \dot{\cup} M$	Set of constructors C and functions M in a μCRL specification.	pp. 25; 26
$\mathcal{A}$	Set of actions in a μCRL specification.	p. 25
$\mathcal{C}$	Set of communication definitions in a μCRL specification.	p. 25
$\mathcal{P}$	Set of processes in a μCRL specification.	p. 25

μCRL and Constraint Solving

$\mathfrak{Z}$	Signature of an algebra.	pp. 25; 22
$\mathcal{S}$	Set of sorts.	pp. 25; 22
$\mathcal{F}$	Set of operation symbols.	pp. 25; 22
σ	Signatures of operations.	pp. 25; 22
$\mathfrak{T}(F,X)$	Set of terms $\mathfrak{T}$ over operations F and variables X.	p. 22
$\mathfrak{A}$	An algebra.	p. 22
A	Carrier set of algebra $\mathfrak{A}$.	p. 22
$\mathfrak{B}$	An algebra for booleans.	p. 23
$\mathfrak{N}$	An algebra for natural numbers.	p. 24
S	Successor symbol for natural numbers.	p. 24
ϑ	An assignment of elements from A to variables.	p. 22
$I^{\vartheta}_{\mathfrak{A}}$	Interpretation function for algebra $\mathfrak{A}$ und assignment function ϑ.	p. 22
$\mathfrak{P}$	A Constraint Logic Program.	p. 36
$\mathfrak{C}$	A constraint.	p. 37
q, q_{init}	(Initial) query to the constraint solver.	p. 36
$\Rightarrow_{LP}$	A resolution relation between two queries.	p. 37
$\mathfrak{T}$	A logical theory.	p. 38
$\mathfrak{D}$	A constraint domain.	p. 38
θ	A solution.	p. 38

Testing with Data Abstraction

Σ_{acc}	Set of accepting states of a test purpose.	p. 60
Σ_{ref}	Set of refusing states of a test purpose.	p. 60
$[\![\mathfrak{M}_{TC}]\!]_{atrace}$	Set of accepted traces of test case $\mathfrak{M}_{TC}$.	p. 63
$[\![\mathfrak{M}_{TC}]\!]_{rtrace}$	Set of refused traces of test case $\mathfrak{M}_{TC}$.	p. 63
$[\![\mathfrak{M}_{TC}]\!]_{Pass}$	Set of traces of test case $\mathfrak{M}_{TC}$, which lead to the verdict Pass.	p. 64
$[\![\mathfrak{M}_{TC}]\!]_{Inconc}$	Set of traces of test case $\mathfrak{M}_{TC}$, which lead to the verdict Inconc.	p. 64
$[\![\mathfrak{M}_{TC}]\!]_{Fail}$	Set of traces of test case $\mathfrak{M}_{TC}$, which lead to the verdict Fail.	p. 64

$\mathfrak{M}^{\mathbb{T}}$	Chaotic abstraction of LTS $\mathfrak{M}$.	p. 65
$\mathbb{S}^{\mathbb{T}}$	The $\mathbb{T}$-abstraction of sort $\mathbb{S}$.	p. 65
$\mathbb{T}_{\mathbb{S}}$	The constant *Chaos* for the sort $\mathbb{S}$.	p. 65
κ	Lifting constructor to lift data instances from an original sort to an abstracted one.	p. 65
γ	May function to map conditions from three-valued logic back to two-valued logic.	p. 66
$\mathfrak{M}_1 \preceq_{\leqslant} \mathfrak{M}_2$	Simulation relation between automata $\mathfrak{M}_1$ and $\mathfrak{M}_2$.	p. 68
β	Already executed subtrace.	p. 75
$\lfloor\theta\rfloor_\beta$	That part of θ, which is relevant for subtrace β; *partial Valuation*.	p. 75
$\mathfrak{O}_\pi$	Oracle for a trace π.	p. 74
$\pi(\theta)$	Trace π under solution θ.	p. 74

Bug Hunting with False Negatives

ζ	Action formula.	p. 151
ϕ, ψ	LTL properties (path formulae).	p. 151
$\Diamond\phi$	ϕ holds in future.	p. 151
$\Box\phi$	ϕ always holds.	p. 151
$\phi\mathbf{U}\psi$	ϕ holds *until* ψ holds.	p. 151
$\phi\mathbf{R}\psi$	ϕ releases ψ.	p. 151
α	General abstraction by an abstraction homomorphism with a state mapping h_s and an action mapping h_a.	p. 152
$\mathfrak{M}^\alpha$, $\alpha(\mathfrak{M})$	Abstraction of LTS $\mathfrak{M}$ (in general and explicitly).	p. 152
$\mathbb{D}^\alpha$	Abstraction data domain for concrete domain $\mathbb{D}$.	p. 153
h_d	A data mapping for abstractions on a symbolic level.	p. 153
k^+	k^+ abstraction for timers with max. concrete value k.	p. 148
Λ_{keep}	Set of non-relaxed actions for a violation pattern.	p. 167
$\mathfrak{V}$	Violation pattern.	p. 167
$\pi_p\pi_s^\omega$	Trace in $\mathfrak{V}$ with stub π_p and (infinite) loop π_s.	p. 167
χ	Concrete counterexample trace.	p. 167
σ_{cyclic}	The first state in the cyclic suffix of $\mathfrak{M}$.	p. 167
p_π	Trace projection function on Λ_{keep}.	p. 168
$\lfloor\pi\rfloor_{\Lambda_{\text{keep}}}$	Projection of trace π on Λ_{keep}	p. 168
$\pi_1 \sim_p \pi_2$	Projection relation between π_1 and π_2.	p. 171

Bibliography

Author References

Stefan C. C. Blom, Jens R. Calamé, Bert Lisser, Simona Orzan, Jun Pang, Jaco van de Pol, Mohammed Torabi Dashti, and Anton J. Wijs. Distributed Analysis with μCRL: A Compendium of Case Studies. In Orna Grumberg and Michael Huth, editors, *Proc. of the 13th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, volume 4424 of *LNCS*, pages 683–689. Springer, 2007. doi:10.1007/978-3-540-71209-1_53.

Jens R. Calamé. Considerations on Object Oriented Software Testing. Preprint 4/2003, Universität Potsdam, Institut für Informatik, 2003.

Jens R. Calamé. Adaptive Test Case Execution in Practice. Technical Report SEN-R0703, Centrum voor Wiskunde en Informatica, 2007.

Jens R. Calamé and Jaco van de Pol. Applying Model-based Testing to HTML Rendering Engines – A Case Study. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *Proc. of the 20th IFIP TC6/WG6.1 Intl. Conf. on Testing of Communicating Systems (TestCom 2008), 7th Intl. Workshop on Formal Approaches to Testing of Software (FATES 2008)*, volume 5047 of *LNCS*, pages 250–265. Springer, 2008. doi:10.1007/978-3-540-68524-1_18.

Jens R. Calamé, Natalia Ioustinova, Jaco van de Pol, and Natalia Sidorova. Data Abstraction and Constraint Solving for Conformance Testing. In Danielle C. Martin, editor, *Proc. of the 12th Asia-Pacific Software Engineering Conf. (APSEC 2005)*, pages 541–548. IEEE Press, 2005. doi:10.1109/APSEC.2005.57.

Jens R. Calamé, Nicolae Goga, Natalia Ioustinova, and Jaco van de Pol. TTCN-3 Testing of Hoorn-Keersenboogerd Railway Interlocking. In *Proc. of the 2006 Canadian Conf. on Electrical and Computer Engineering (CCECE 2006)*, pages 620–623. IEEE Press, 2006a. doi:10.1109/CCECE.2006.277762.

Jens R. Calamé, Natalia Ioustinova, and Jaco van de Pol. Automatisierte Erzeugung von TTCN-3 Testfällen aus UML-Modellen. In Christian Hochberger and Rüdiger Liskowsky, editors, *Informatik 2006*, Lecture Notes in Informatics, pages 257–261. Gesellschaft für Informatik, October 2006b.

Jens R. Calamé, Natalia Ioustinova, and Jaco van de Pol. Automatic Model-Based Generation of Parameterized Test Cases Using Data Abstraction. In Judi Romijn, Graeme Smith, and Jaco van de Pol, editors, *Proc. of the Doctoral Symp. affiliated with the 5th Intl. Conf. on Integrated Formal Methods (IFM 2005)*, volume 191

of *Electronic Notes in Theoretical Computer Science*, pages 25–48. Elsevier, 2007a. doi:10.1016/j.entcs.2007.06.019.

Jens R. Calamé, Natalia Ioustinova, Jaco van de Pol, and Natalia Sidorova. Bug Hunting with False Negatives. In Jim Davies and Jeremy Gibbons, editors, *Proc. of the 6th Intl. Conf. on Integrated Formal Methods (IFM 2007)*, volume 4591 of *LNCS*, pages 98–117. Springer, 2007b. doi:10.1007/978-3-540-73210-5_6.

Other References

Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer, and Franz Wotawa. Test Purpose Generation in an Industrial Application. In *Proc. of the 3rd Intl. Workshop on Advances in Model-based Testing*, pages 115–125. ACM Press, 2007. doi:10.1145/1291535.1291547.

Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming Using Eclipse*. Cambridge University Press, 2007.

Paul Baker, Zhen Ru Dai, Jens Grabowski, Øystein Haugen, Ina Schieferdecker, and Clay Williams. *Model-Driven Testing – Using the UML Testing Profile*. Springer, 2008.

Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proc. of the 4th Intl. Conf. on Integrated Formal Methods (IFM 2004)*, volume 2999 of *LNCS*, pages 1–20. Springer, 2004. doi:10.1007/b96106.

Kent Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.

Bert van Beek, Ka L. Man, Michel A. Reniers, Koos E. Rooda, and Ramon R. H. Schiffelers. Syntax and Consistent Equation Semantics of Hybrid Chi. *Journal of Logic and Algebraic Programming*, 68(1–2):129–210, 2006. doi:10.1016/j.jlap.2005.10.005.

Axel Belinfante, Jan Feenstra, René de Vries, Jan Tretmans, Nicolae Goga, Loe Feijs, Sjouke Mauw, and Lex Heerink. Formal Test Automation: A Simple Experiment. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *12th Intl. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.

Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre McKenzie. *Systems and Software Verification*. Springer, 2001.

Jan A. Bergstra, Jan W. Klop, and Aart Middeldorp. *Termherschrijfsystemen*. Kluwer Bedrijfswetenschappen, 1989.

Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

Stefan C. C. Blom, Natalia Ioustinova, and Natalia Sidorova. Timed Verification with μCRL. In Manfred Broy and Alexandre V. Zamulin, editors, *Proc. of the 5th Intl. Conf. on Perspectives of System Informatics (PSI 2003)*, volume 2890 of *LNCS*, pages 178–192. Springer, 2003. doi:10.1007/b94823.

Barry W. Boehm. Guidelines for Verifying and Validating Software Requirements and Design Specifications. In *Proc. of the Euro IFIP 1979*, pages 711–719, 1979.

Marc Born, Hans-Gerard Gross, Pedro Santos, and Ina Schieferdecker. Model-driven Development and Testing – A Case Study. In Marten J. van Sinderen and Luís Ferreira Pires, editors, *1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, number TR-CTIT-04-12 in CTIT Technical Report, pages 97–104, Enschede, 2004.

Wiet Bouma. *Algebraïsche Specificaties*. Kluwer Programmatuurkunde, 1991.

Laura Brandán Briones. *Theories for Model-Based Testing: Real-time and Coverage*. PhD thesis, Universiteit Twente, 2007. URL `http://doc.utwente.nl/57810/`.

Laura Brandán Briones and Ed Brinksma. A Test Generation Framework for quiescent Real-Time Systems. In Jens Grabowski and Brian Nielsen, editors, *Proc. of the 4th Intl. Workshop on Formal Approaches to Testing (FATES 2004)*, volume 3395 of *LNCS*, pages 64–78. Springer, 2005. doi:10.1007/b106767.

Lionel C. Briand and Yvan Labiche. A UML-Based Approach to System Testing. In Martin Gogolla and Cris Kobryn, editors, *Proc. of the 4th Intl. Conf. on The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML 2001)*, volume 2185 of *LNCS*, pages 194–208. Springer, 2001. ISBN 3-540-42667-1. doi:10.1007/3-540-45441-1_15.

BRICKS. Basic Research in Informatics for Creating the Knowledge Society. URL `http://www.bsik-bricks.nl`.

Pascal Brisset et al. *ECLiPSe Constraint Library Manual*, version 5.9 edition, 2006.

Manfred Broy. Compositional Refinement of Interactive Systems Modelled by Relations. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference (COMPOS 1997)*, volume 1536 of *LNCS*, pages 130–149. Springer, 1998. doi:10.1007/3-540-49213-5_6.

Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2005. doi:10.1007/b137241.

Ingo Brückner, Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim. Slicing Abstractions. In Farhad Arbab and Marjan Sirjani, editors, *Proc. of the Intl. Symp. on Fundamentals of Software Engineering (FSEN 2007)*, volume 4767 of *LNCS*, pages 17–32. Springer, 2007. doi:10.1007/978-3-540-75698-9_2.

CEPSCO. *Common Electronic Purse Specifications, Functional Requirements*. CEPSCO, 1999. Version 6.3.

CEPSCO. *Common Electronic Purse Specifications, Technical Specification*. CEPSCO, 2000. Version 2.2.

Sagar Chaki, Edmund Clarke, Orna Grumberg, Joël Ouaknine, Natasha Sharygina, Tayssir Touili, and Helmut Veith. State/Event Software Verification for Branching-Time Specifications. In Judi Romijn, Graeme Smith, and Jaco van de Pol, editors, *Proc. of the 5th Intl. Conf. on Integrated Formal Methods (IFM 2005)*, volume 3771 of *LNCS*, pages 53–69. Springer, 2005. doi:10.1007/11589976_5.

Kwang Ting Cheng and A. S. Krishnakumar. Automatic Functional Test Generation Using the Extended Finite State Machine Model. In *Proc. of the 30th Intl. Conf. on Design Automation (DAC 1993)*, pages 86–91. ACM, 1993. ISBN 0-89791-577-1. doi:10.1145/157485.164585.

Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Automated Test and Oracle Generation for Smart-Card Applications. In Isabelle Attali and Thomas Jensen, editors, *Proc. of the Intl. Conf. on Research in Smart Cards (e-Smart 2001)*, volume 2140 of *LNCS*, pages 58–70. Springer, 2001. doi:10.1007/3-540-45418-7_6.

Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A Symbolic Test Generation Tool. In Joost-Pieter Katoen und Perdita Stevens, editor, *Proc. of the 8th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 470–475. Springer, 2002. doi:10.1007/3-540-46002-0_34.

Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994. doi:10.1145/186025.186051.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *Journ. of the ACM*, 50(5):752–794, 2003. doi:10.1145/876638.876643.

William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer, fourth edition, 1994.

Conformiq. *Conformiq Qtronic – Quick Start, Installation, Use, Reference*.

Camille Constant, Thierry Jéron, Hervé Marchand, and Vlad Rusu. Integrating Formal Verification and Conformance Testing for Reactive Systems. *IEEE Transactions on Software Engineering*, 33(8):558–574, 2007. doi:10.1109/TSE.2007.70707.

Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of programming languages (POPL 1977)*, pages 238–252. ACM Press, 1977. doi:10.1145/512950.512973.

Dennis Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Technische Universiteit Eindhoven, 1996.

Dennis Dams and Rob Gerth. The Bounded Retransmission Protocol Revisited. In Faron Moller, editor, *Proc. of the 2nd Intl. Workshop on Verification of Infinite State Systems (Infinity 1997)*, volume 9 of *Electronic Notes in Theoretical Computer Science*, page 26. Elsevier, 1999. doi:10.1016/S1571-0661(05)80425-6.

Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997. doi:10.1145/244795.244800.

Satyaki Das and David L. Dill. Counter-Example Based Predicate Discovery in Predicate Abstraction. In Mark D. Aagaard and John W. O'Leary, editors, *Proc. of the 4th Intl. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2002)*, volume 2517 of *LNCS*, pages 19–32. Springer, 2002. doi:10.1007/3-540-36126-X_2.

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Proc. of the 21st Intl. Conf. on Software Engineering (ICSE 1999)*, pages 411–420. IEEE Press, 1999. doi:10.1109/ICSE.1999.841031.

Marcel Erné, Jürgen Koslowski, Austin Melton, and George E. Strecker. A primer on galois connections. *Annals of the New York Academy of Sciences*, 704:103–125, 1993.

ETSI. ETSI ES 201 873-1 V2.2.1: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. ETSI Standard, 2003a.

ETSI. ETSI ES 201 873-6 V1.1.1: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI). ETSI Standard, 2003b.

ETSI. ETSI ES 201 873-5 V1.1.1: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI). ETSI Standard, 2003c.

Wan Fokkink. *Introduction to Process Algebra*. EATCS. Springer, 2000.

Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test Generation Based on Symbolic Specifications. In Jens Grabowski and Brian Nielsen, editors, *FATES 2004*, volume 3395 of *LNCS*, pages 1–15. Springer, 2005. doi:10.1007/b106767.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

Hubert Garavel and Frédéric Lang. NTIF: A General Symbolic Model for Communicating Sequential Processes with Data. In Doron A. Peled and Moshe Y. Vardi, editors, *Proc. of the 22nd IFIP WG 6.1 Intl. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE 2002)*, volume 2529 of *LNCS*, pages 276–291. Springer, 2002. doi:10.1007/3-540-36135-9_18.

Angelo Gargantini. Conformance Testing. In Broy et al. (2005). doi:10.1007/11498490_5.

David Gelperin and Bill Hetzel. The Growth of Software Testing. *Communications of the ACM*, 31(6):687–695, 1988. doi:10.1145/62959.62965.

Wouter Geurts, Klaas Wijbrans, and Jan Tretmans. Testing and Formal Methods – Bos Project Case Study. In *Proc. of the 6th European Intl. Conf. on Software Testing, Analysis & Review (EuroSTAR 1998)*, pages 215–229, 1998. URL `http://eprints.eemcs.utwente.nl/6495/`.

Dimitra Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, University of London, 1999.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'05)*, pages 213–223, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-056-6. doi:http://doi.acm.org/10.1145/1065010.1065036.

Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An Introduction to the Testing and Test Control Notation (TTCN-3). *Computer Networks*, 42(3):375–403, 2003. doi:10.1016/S1389-1286(03)00249-4.

Jan Friso Groote. The Syntax and Semantics of Timed μCRL. Technical Report SEN-R9709, Centrum voor Wiskunde en Informatica, 1997.

Jan Friso Groote and Alban Ponse. The Syntax and Semantics of μCRL. In Alban Ponse, Chris Verhoef, and Bas van Vlijmen, editors, *Algebra of Communicating Processes*, Workshops in Computing, pages 26–62. Springer, 1994.

Orna Grumberg, Flavio Lerda, Ofer Strichman, and Michael Theobald. Proof-guided Underapproximation-Widening for Multi-Process Systems. In *Proc. of the 32nd ACM SIGACT-SIGPLAN Symp. on Principles of programming languages (POPL 2005)*, pages 122–131. ACM Press, 2005. doi:10.1145/1040305.1040316.

Hans W. Guesgen. Constraints. In Günther Görz, Claus-Rainer Rollinger, and Josef Schneeberger, editors, *Handbuch der Künstlichen Intelligenz*, chapter 8, pages 267–287. Oldenbourg, third edition, 2000.

David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987. doi:10.1016/0167-6423(87)90035-9.

Rob Hendriks, Erik van Veenendaal, and Robert van Vonderen. Measuring Software Quality. In Veenendaal (2002), chapter 6, pages 91–102.

Robert M. Hierons. Applying Adaptive Test Cases to Nondeterministic Implementations. *Information Processing Letters*, 98:56–60, 2006. doi:10.1016/j.ipl.2005.12.001.

David Hoyle. *ISO 9000 Quality Systems Handbook*. Butterworth-Heinemann, fifth edition, 2005.

Thomas Ihringer. *Allgemeine Algebra*. B. G. Teubner, second edition, 1993.

Natalia Ioustinova. *Abstractions and Static Analysis for Verifying Reactive Systems*. PhD thesis, Vrije Universiteit Amsterdam, 2004. URL `http://hdl.handle.net/1871/9061`.

Natalia Ioustinova, Natalia Sidorova, and Martin Steffen. Abstraction and Flow Analysis for Model Checking Open Asynchronous Systems. In *Proc. of the 9th Asia-Pacific Software Engineering Conf. (APSEC 2002)*, pages 227–235. IEEE Press, 2002a. doi:10.1109/APSEC.2002.1182992.

Natalia Ioustinova, Natalia Sidorova, and Martin Steffen. Closing Open SDL-Systems for Model Checking with DTSpin. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods - Getting IT Right, Proc. of the Intl. Symp. of Formal Methods Europe (FME 2002)*, volume 2391 of *LNCS*, pages 157–177. Springer, 2002b. doi:10.1007/3-540-45614-7_30.

Natalia Ioustinova, Natalia Sidorova, and Martin Steffen. Synchronous Closing and Flow Abstraction for Model Checking Timed Systems. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proc. of the 2nd Intl. Symp. on Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *LNCS*, pages 292–313. Springer, 2004. doi:10.1007/b100112.

ISTQB. Standard Glossary of Terms Used in Software Testing, June 2006. URL `http://www.istqb.org/downloads/glossary-1.2.pdf`. Version 1.2 final.

ITU-T. Recommendation X.290-ISO/IEC 9646-1, Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 1: General Concepts, 1996. URL `http://www.itu.int/rec/T-REC-X.290/en`.

ITU-T. Recommendation Z.120, Message Sequence Chart, 2005. URL `http://www.itu.int/rec/T-REC-Z.120/en`.

Claude Jard and Thierry Jéron. TGV: Theory, Principles and Algorithms. *Intl. Journ. on Software Tools for Technology Transfer*, 7(4):297–315, 2005. doi:10.1007/s10009-004-0153-x.

Bertrand Jeannet, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Symbolic Test Selection based on Approximate Analysis. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Proc. of the 11th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, number 3440 in LNCS, pages 349–364. Springer, 2005. doi:10.1007/b107194.

Thierry Jéron. *Contribution à la génération automatique des tests pour les systèmes réactifs*. Habilitation thesis, L'Université de Rennes 1, 2004.

Jan Jürjens. *Secure Systems Development with UML*. Springer, Berlin, Heidelberg, 2005.

Jan Jürjens and Guido Wimmel. Security Modelling for Electronic Commerce: The Common Electronic Purse Specifications. In Beat Schmid, Katarina Stanoevska-Slabeva, and Volker Tschammer, editors, *Towards the E-Society. Proc. of the 1st IFIP Intl. Conf. on E-Commerce, E-Business and E-Government*, pages 489 – 506. Kluwer Academic Publishers, 2001.

Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, second edition, 2003.

Yonit Kesten and Amir Pnueli. Control and Data Abstraction: The Cornerstones of Practical Formal Verification. *Intl. Journ. on Software Tools for Technology Transfer*, 2 (4):328–342, 2000. doi:10.1007/s100090050040.

Moez Krichen and Stavros Tripakis. Black-Box Conformance Testing for Real-Time Systems. In Susanne Graf and Laurent Mounier, editors, *Proc. of the 11th Intl. SPIN Workshop (ICTAC 2006)*, volume 2989 of *LNCS*, pages 109–126. Springer, 2004. doi:10.1007/b96721.

Moez Krichen and Stavros Tripakis. Interesting Properties of the Real-Time Conformance Relation tioco. In Kamel Barkaoui, Ana Cavalcanti, and Antonio Cerone, editors, *Proc. of the 3rd Intl. Colloquium on Theoretical Aspects of Computing (ICTAC 2006)*, volume 4281 of *LNCS*, pages 317–331. Springer, 2006. doi:10.1007/11921240_22.

Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, third edition, 2003.

Yassine Lakhnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental Verification by Abstraction. In Tiziana Margaria and Wang Yi, editors, *Proc. of the 7th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 98–112. Springer, 2001. doi:10.1007/3-540-45319-9_8.

Kim G. Larsen and Bent Thomsen. A Modal Process Logic. In *Proc. of the Third Annual Symp. on Logic in Computer Science (LICS 1988)*, pages 203–210. IEEE Press, 1988. doi:10.1109/LICS.1988.5119.

Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online Testing of Real-time Systems Using Uppaal. In Jens Grabowski and Brian Nielsen, editors, *Proc. of the 4th Intl. Workshop on Formal Approaches to Software Testing (FATES 2004)*, volume 3395 of *LNCS*, pages 79–94. Springer, 2005. doi:10.1007/b106767.

Yves Ledru, Pierre Bontron, Lydie du Bousquet, Olivier Maury, and Catherine Oriat. TOBIAS : un outil de test combinatoire pour le test de conformité. In *Acte de conférence de la quatorzième édition des Journées Internationales "Génie Logiciel & Ingénierie de Systèmes et leurs Applications" (ICSSEA 2001)*, 2001.

Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6(1):11–44, 1995. doi:10.1007/BF01384313.

Heiko Lötzbeyer and Alexander Pretschner. AutoFocus on Constraint Logic Programming. In *Proc. of the (Constraint) Logic Programming and Software Engineering (LPSE 2000)*, 2000.

Nancy A. Lynch and Mark R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proc. of the 6th Annual ACM Symp. on Principles of Distributed Computing (PODC 1987)*, pages 137–151. ACM Press, 1987. doi:10.1145/41840.41852.

Steve Maguire. *Nie wieder Bugs*. Fachbibliothek Programmierung. Microsoft Press, 1998.

Kim Marriott and Peter J. Stuckey. *Programming with Constraints – An Introduction*. MIT Press, 1998.

Augustus de Morgan. *Syllabus of a Proposed System of Logic*. Walton and Maberly, 1860.

Glenford J. Myers. *The Art of Software Testing*. Wiley, 1979.

Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal Strategies for Testing Nondeterministic Systems. In *Proc. of the 2004 ACM SIGSOFT Intl. Symp. on Software Testing and Analysis (ISSTA 2004)*, pages 55–64. ACM Press, 2004. doi:10.1145/1007512.1007520.

Peter Naur and Brian Randell, editors. *Software Engineering – Report on a Conference sponsored by the NATO SCIENCE COMMITTEE*, 1969. NATO.

Rocco de Nicola and Matthew Hennessy. Testing Equivalence for Processes. In Josep Diaz, editor, *Proc. of the 10th Colloquium on Automata, Languages and Programming*, volume 154 of *LNCS*, pages 548–560. Springer, 1983. doi:10.1007/BFb0036936.

Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Guided Test Generation from CSP Models. Technical report, Centro de Informática, Universidade Federal de Pernambuco, Brazil, 2007.

Stefano Novello, Joachim Schimpf, Kish Shen, and Josh Singer. *ECLiPSe Embedding and Interfacing Manual*, version 5.8 edition, 2005.

Jeff Offutt and Aynur Abdurazik. Generating Tests from UML Specifications. In Robert France and Bernhard Rumpe, editors, *Proc. of the 2nd Intl. Conf. on The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML 1999)*, volume 1723 of *LNCS*, pages 416–429. Springer, 1999. doi:10.1007/3-540-46852-8_30.

OMG. UML 2.0 Testing Profile Specification, 2003.

OMG. UML 2.0 Superstructure Specification, 2005.

Gordon Pace, Nicolas Halbwachs, and Pascal Raymond. Counter-example Generation in Symbolic Abstract Model-Checking. *Intl. Journ. on Software Tools for Technology Transfer*, 5(2):158–164, 2004. ISSN 1433-2779. doi:10.1007/s10009-003-0127-4.

David Park. Concurrency and Automata on Infinite Sequences. In G. Goos and J. Hartmanis, editors, *Proc. of the 5th GI Conf.*, volume 104 of *LNCS*, pages 167–183. Springer, 1981. doi:10.1007/BFb0017309.

Carl A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Bonn, 1962.

Jaco van de Pol and Miguel A. Valero Espada. Modal Abstractions in μCRL. In Charles Rattray, Savitri Maharaj, and Carron Shankland, editors, *Proc. of the 10th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST 2004)*, volume 3116 of *LNCS*, pages 409–425. Springer, 2004. doi:10.1007/b98770.

Alexander Pretschner, Heiko Lötzbeyer, and Jan Philipps. Model-based Testing in Incremental System Development. *Journ. of Systems and Software*, 70(3):315–329, 2004a. ISSN 0164-1212. doi:10.1016/S0164-1212(03)00076-1.

Alexander Pretschner, Oscar Slotosch, E. Aiglstorfer, and Stefan Kriebel. Model-based Testing for Real. *Intl. Journ. on Software Tools for Technology Transfer*, 5(2–3): 140–157, 2004b. doi:10.1007/s10009-003-0128-3.

Corina S. Păsăreanu, Matthew B. Dwyer, and Willem Visser. Finding Feasible Counter-examples when Model Checking Abstracted Java Programs. In Tiziana Margaria and Wang Yi, editors, *Proc. of the 7th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 284–298. Springer, 2001. doi:10.1007/3-540-45319-9_20.

Corina S. Păsăreanu, Radek Pelánek, and Willem Visser. Concrete Model Checking with Abstract Matching and Refinement. In Kousha Etessami und Sriram K. Rajamani, editor, *Proc. of the 17th Intl. Conf. on Computer-Aided Verification (CAV 2005)*, volume 3576 of *LNCS*, pages 52–66. Springer, 2005. doi:10.1007/11513988_7.

J. Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journ. of the ACM*, 12(1):23–41, 1965. doi:10.1145/321250.321253.

Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An Approach to Symbolic Test Generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Proc. of the 2nd Intl. Conf. on Integrated Formal Methods (IFM 2000)*, volume 1945 of *LNCS*, pages 338–357. Springer, 2000. doi:10.1007/3-540-40911-4_20.

Roberto Segala. Quiescence, Fairness, Testing, and the Notion of Implementation. In Eike Best, editor, *Proc. of the 4th Intl. Conf. on Concurrency Theory (CONCUR 1993)*, volume 715 of *LNCS*, pages 324–338. Springer, 1993. doi:10.1007/3-540-57208-2_23.

Natalia Sidorova and Martin Steffen. Verifying Large SDL-Specifications Using Model Checking. In Rick Reed and Jeanne Reed, editors, *Proc. of the 10th Intl. SDL Forum (SDL 2001)*, volume 2078 of *LNCS*, pages 403–420. Springer, 2001a. doi:10.1007/3-540-48213-X_25.

Natalia Sidorova and Martin Steffen. Embedding Chaos. In Patrick Cousot, editor, *Proc. of the 8th Intl. Static Analysis Symp. (SAS 2001)*, volume 2126 of *LNCS*, pages 319–334. Springer, 2001b. doi:10.1007/3-540-47764-0_18.

Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1981.

Jan Tretmans. Test Generation with Inputs, Outputs, and Repetitive Quiescence. *Software – Concepts & Tools*, 17(3):103–120, 1996.

Jan Tretmans and Ed Brinksma. TorX: Automated Model-based Testing. In Alan Hartman and Klaudia Dussa-Ziegler, editors, *Proc. of the 1st European Conf. on Model-Driven Software Engineering*, 2003.

TT-Medal. Tests & Testing Methodologies for Advanced Languages. online: `http://www.tt-medal.org`.

Doug Turner and Ian Oeschger. *Creating XPCOM Components*, 2003. online.

Yaroslav S. Usenko. *Linearization in μCRL*. PhD thesis, Technische Universiteit Eindhoven, 2002.

Frits W. Vaandrager. On the Relationship Between Process Algebra and Input/Output Automata. In *Proc. of the 6th Annual Symp. on Logic in Computer Science (LICS 1991)*, pages 387–398. IEEE Press, 1991. Extended abstract.

Erik van Veenendaal, editor. *The Testing Practitioner*. Uitgeverij Tutein Nolthenius, 2002.

W3C. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition), 2002. URL `http://www.w3.org/TR/2002/REC-xhtml1-20020801`. Recommendation.

W3C. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, 2007. URL `http://www.w3.org/TR/2007/CR-CSS21-20070719`. Candidate Recommendation.

Anton J. Wijs. Achieving Discrete Relative Timing with Untimed Process Algebra. In *Proc. of the 12th IEEE Intl. Conf. on Engineering Complex Computer Systems (ICECCS 2007)*, pages 35–46. IEEE Press, 2007. doi:10.1109/ICECCS.2007.13.

Colin Willcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Frederico Engler, and Stephan Schulz. *An Introduction to TTCN-3*. Wiley, 2005.

Nicholas C. Zakas, Jeremy McPeak, and Joe Fawcett. *Professional AJAX*. Wrox Press, 2006.

Eléna Zinovieva-Leroux. *Méthodes symboliques pour la génération de tests de systèmes reactifs comportant des données*. PhD thesis, Université de Rennes, 2004.

Index

Summary

Software faults are a well-known phenomenon. In most cases, they are just annoying – if the computer game does not work as expected – or expensive – if once again a space project fails due to some faulty data conversion. In critical systems, however, faults can have life-threatening consequences. It is the task of software quality assurance to avoid such faults, but this is a cumbersome, expensive and also erroneous undertaking. For this reason, research has been done over the last years in order to automate this task as much as possible.

In this thesis, the connection of constraint solving techniques with formal methods is investigated. We have the goal to find faults in the models and implementations of reactive systems with data, such as automatic teller machines (ATMs). In order to do so, we first develop a translation of formal specifications in the process algebra μCRL to a constraint logic program (CLP). In the course of this translation, we pay special attention on the fact that the CLP together with the constraint solver correctly simulates the underlying term rewriting system.

One way to validate a system is the test whether this system conforms its specification. In this thesis, we develop a test process to automatically generate and execute test cases for the conformance test of data-oriented systems. The applicability of this process to process-oriented software systems is demonstrated in a case study with an ATM as the system under test. The applicability of the process to document-centered applications is shown by means of the open source web browser Mozilla Firefox.

The test process is partially based on the tool TGV, which is an enumerative test case generator. It generates test cases from a system specification and a test purpose. An enumerative approach to the analysis of system specifications always tries to enumerate all possible combinations of values for the system's data elements, i.e. the system's states. The states of those systems, which we regard here, are influenced by data of possibly infinite domains. Hence, the state space of such systems grows beyond all limits, it explodes, and cannot be handled anymore by enumerative algorithms. For this reason, the state space is limited prior to test case generation by a data abstraction. We use a *chaotic* abstraction here with all possible input data from a system's environment being replaced by a single constant.

In parallel, we generate a CLP from the system specification. With this CLP, we reintroduce the actual data at the time of test execution. This approach does not only limit the state space of the system, but also leads to a separation of system behavior and data. This allows to reuse test cases by only varying their data parameters.

In the developed process, tests are executed by the tool BAiT. This tool has also been created in the course of this thesis. Some systems do not always show an identical behavior under the same circumstances. This phenomenon is known as *nondeterminism*. There are many reasons for nondeterminism. In most cases, input from

a system's environment is asynchronously processed by several components of the system, which do not always terminate in the same order. BAiT works as follows: The tool chooses a trace through the system behavior from the set of traces in the generated test cases. Then, it parameterizes this trace with data and tries to execute it. When the nondeterministic system digresses from the selected trace, BAiT tries to appropriately adapt it. If this can be done according to the system specification, the test can be executed further and a possibly false positive test verdict has been successfully avoided.

The test of an implementation significantly reduces the numbers of faults in a system. However, the system is only tested against its specification. In many cases, this specification already does not completely fulfill a customer's expectations. In order to reduce the risk for faults further, the models of the system themselves also have to be verified. This happens during model checking prior to testing the software. Again, the explosion of the state space of the system must be avoided by a suitable abstraction of the models.

A consequence of model abstractions in the context of model checking are so-called *false negatives*. Those traces are counterexamples which point out a fault in the abstracted model, but who do not exist in the concrete one. Usually, these false negatives are ignored. In this thesis, we also develop a methodology to reuse the knowledge of potential faults by abstracting the counterexamples further and deriving a *violation pattern* from it. Afterwards, we search for a concrete counterexample utilizing a constraint solver.

Samenvatting

Testen van reactieve systemen met gegevens
Opsommende methoden en constraint solving

Fouten in software zijn een welbekend fenomeen. Meestal alleen maar lastig – als het computerspelletje het niet goed doet – of duur – als er weer een ambitieus ruimtevaartproject zijn abrupt einde vindt in een foute dataconversie –, kunnen zij in kritieke systemen ook levensbedreigende gevolgen hebben. Het is de taak van de softwarekwaliteitsbewaking om deze fouten te voorkomen, maar dit is dikwijls een tijdrovende, dure en zelf ook weer foutgevoelige onderneming. Daarom werd er in de laatste jaren van verschillende kanten onderzoek verricht, om deze taak zo ver mogelijk te automatiseren.

In dit proefschrift wordt de verbinding van constraint solving technieken met formele methoden behandeld, met het doel, fouten in de modellen en de implementatie van reactieve systemen met gegevens, zoals bv. geldautomaten, op te sporen. Hiervoor wordt een vertaling van een formele specificatie in de procesalgebra μCRL naar een constraint programma (CLP) ontwikkeld. In het kader van deze vertaling wordt er met name op gelet, dat de verwerking van het CLP door de constraint solver een correcte simulatie van het termherschrijfsysteem is, waarop dit CLP gebaseerd is.

Een manier van systeemvalidatie is het testen op de conformiteit van een systeem ten opzichte van zijn specificatie. In dit proefschrift wordt onder meer een testproces ontworpen, om testgevallen voor de conformiteitstest van de bovengenoemde datageoriëntieerde systemen geautomatiseerd te kunnen genereren en uitvoeren. De toepasselijkheid van dit proces voor procesgeoriënteerde softwaresystemen wordt met een casestudy, met als te testen systeem een geldautomaat, getoond. Zijn toepasselijkheid voor documentgeoriënteerde systemen wordt aan de hand van de open-source webbrowser Mozilla Firefox geverifieerd.

Het testproces is gedeeltelijk gebaseerd op het tool TGV, een optellende generator voor testgevallen vanuit de systeemspecificatie en een testdoel, de zogenoemde *test purpose*. Een optellende aanpak voor de analyse van systeemspecificaties probeert altijd alle mogelijke combinaties van waarden van data-elementen in het systeem, diens toestanden dus, op te tellen. De toestanden van de systemen, welke wij hier bestuderen, zijn beïnvloed door gegevens van mogelijk onbeperkte domeinen. Dientengevolge groeit de toestandsruimte van een dergelijk systeem over alle grenzen, zodat een optellende aanpak niet geschikt is. Om die reden wordt de toestandsruimte voorafgaand aan de testgeneratie door een data-abstractie beperkt. Hiervoor gebruiken wij een *chaotische* abstractie, waarbij alle mogelijke invoergegevens van buiten het systeem door een enkele constante worden vervangen.

Tegelijk wordt een CLP van de specificatie gegenereerd, met wiens hulp de eigenli-

jke gegevens tijdens de testuitvoering weer ingevoerd worden. Deze aanpak beperkt niet alleen de toestandsruimte van het systeem tijdens de testgeneratie, maar stelt ook een scheiding van het systeemgedrag en zijn gegevens voor. Dit maakt het mogelijk, een testgeval meerdere keren te hergebruiken door alleen zijn parameters anders te bepalen.

De tests worden volgens dit proces door het tool BAiT uitgevoerd, dat in het kader van dit proefschrift ontstaan is. Sommige systemen tonen niet altijd hetzelfde gedrag onder dezelfde omstandigheden, een fenomeen dat als *nondeterminisme* bekend staat. De redenen daarvoor zijn veelvoudig; zo kan een systeem bijvoorbeeld meerdere componenten bevatten, die een invoer uit hun omgeving gelijkertijd bewerken en er niet altijd in dezelfde volgorde mee klaar zijn. BAiT werkt als volgt: Het tool kiest een pad door het systeemgedrag uit een verzameling van paden in de gegenereerde testgevallen, bepaalt zijn parameters en probeert hem draaien. Zodra het nondeterministische systeem van dit pad afwijkt, probeert BAiT de testuitvoering geschikt aan te passen. Als dit volgens de systeemspecificatie mogelijk is, kan de test doorgaan en is een mogelijk foutief oordeel vermeden.

De test van een implementatie vermindert de kans op fouten aanzienlijk; het systeem wordt niettemin alleen ten opzichte van zijn specificatie gecontrolleerd. In veel gevallen voldoet deze specificatie al niet volledig aan de eisen van de opdrachtgever. Om het risico op fouten nog verder terug te dringen, moeten dus de modellen van de software zelf ook geverifieerd worden. Dit gebeurt tijdens een modelverificatiefase, die in het software-ontwikkelingsproces nog voor het testen plaats vindt. Ook hier moet weer de explosie van de toestandsruimte worden voorkomen door het model te abstraheren.

Een gevolg van modelabstracties in het kader van de modelverficatie zijn zogenoemde *false negatives*, aangetoonde tegenvoorbeelden met fouten in het geabstraheerde model, die in het originele model niet teruggevonden kunnen worden. Normaliter worden deze false negatives genegeerd, maar in dit proefschrift wordt een methode ontwikkeld om ook van deze bevindingen gebruik te kunnen maken. Daarvoor wordt het tegenvoorbeeld verder geabstraheerd en een schendingspatroon afgeleid. Met behulp van een constraint solver wordt dan een ècht tegenvoorbeeld in het systeem gezocht.

Zusammenfassung

Test reaktiver Systeme mit Daten
Aufzählende Methoden und Constraint-Solving

Softwarefehler sind ein wohlbekanntes Phänomen. Zumeist lediglich lästig – wenn etwa ein Computerspiel nicht richtig funktioniert – oder teuer – wenn wieder einmal ein Raumfahrtprojekt sein jähes Ende in einer fehlerhaften Datenkonvertierung findet –, können sie bei kritischen Systemen durchaus lebensbedrohliche Konsequenzen haben. Es ist die Aufgabe der Softwarequalitätssicherung, diese Fehler zu vermeiden, jedoch ist dies häufig ein zeitraubendes, teures und auch selbst fehlerträchtiges Unterfangen. Darum wurde in den vergangenen Jahren von verschiedenen Seiten daran geforscht, diese Aufgabe soweit möglich zu automatisieren.

In dieser Dissertation wird die Verbindung von Constraint-Solving-Techniken mit Formalen Methoden unter der Zielsetzung behandelt, Fehler in den Modellen und der Implementation von reaktiven Systemen mit Daten, wie z.B. Geldautomaten, aufzuspüren. Hierfür wird zunächst eine Übersetzung einer formalen Spezifikation in der Prozessalgebra μCRL in ein Constraint-Programm (CLP) entwickelt. Im Rahmen dieser Übersetzung wird insbesondere darauf geachtet, dass die Bearbeitung des CLP durch einen Constraint-Solver eine korrekte Simulation des Term-Rewriting-Systems ist, auf dem das CLP basiert.

Eine Art der Systemvalidierung ist der Test auf Konformität eines Systems hinsichtlich seiner Spezifikation. In dieser Dissertation wird unter anderem ein Testprozess entworfen, mit dem Testfälle für den Konformitätstest der oben genannten datenorientierten Systeme automatisiert generiert und ausgeführt werden können. Die Anwendbarkeit dieses Prozesses für prozessorientierte Softwaresysteme wird mit einer Fallstudie mit einem Geldautomaten als zu testendem System nachgewiesen. Seine Anwendbarkeit auf dokumentenorientierte Systeme wird anhand des quelloffenen Webbrowsers Mozilla Firefox verifiziert.

Der Testprozess basiert teilweise auf dem Werkzeug TGV, einem enumerativen Generator für Testfälle aus einer Systemspezifikation und einem Testziel, einem sogenannten *Test Purpose*. Eine enumerative Herangehensweise für die Analyse von Systemspezifikationen probiert immer, alle möglichen Wertekombinationen von Datenelementen im System, seine Zuständen also, aufzuzählen. Die Zustände jener Systeme, die wir hier betrachten, werden durch Daten möglicherweise unendlicher Wertebereiche beeinflusst. Demzufolge wächst auch der Zustandsraum derartiger Systeme über alle Grenzen, sodass er von enumerativen Algorithmen nicht mehr zu erfassen ist. Aus diesem Grunde wird der Zustandsraum vor der Testfallgenerierung durch eine Datenabstraktion beschränkt. Hierfür verwenden wir eine *chaotische* Abstraktion, wobei alle möglichen Eingabedaten von außerhalb des Systems durch eine

einzelne Konstante ersetzt werden.

Zugleich wird aus der Spezifikation des Systems ein CLP generiert, mit dessen Hilfe die eigentlichen Daten zum Zeitpunkt der Testausführung wieder eingeführt werden. Diese Herangehensweise beschränkt nicht nur den Zustandsraum des Systems, sondern stellt auch eine Trennung zwischen Systemverhalten und -daten dar. Dies ermöglicht eine Wiederverwendung von Testfällen durch variierende Parametrisierung.

Im entwickelten Prozess werden Tests durch das Werkzeug BAiT ausgeführt, das ebenfalls im Rahmen dieser Dissertation entstanden ist. Manche Systeme zeigen nicht immer dasselbe Verhalten unter denselben Umständen, ein Phänomen das als *Nichtdeterminismus* bekannt ist. Die Gründe dafür sind vielfältig, so kann ein System beispielsweise aus mehreren Komponenten bestehen, die eine Eingabe aus der Systemumgebung gleichzeitig verarbeiten, diese Verarbeitung jedoch nicht immer in derselben Reihenfolge abschließen. BAiT funktioniert wie folgt: Das Werkzeug wählt einen Pfad durch das Systemverhalten aus einer Menge von Pfaden in den generierten Testfällen aus, parametrisiert diesen und probiert, ihn auszuführen. Sobald das nichtdeterministische System von diesem Pfad abweicht, probiert BAiT, diesen geeignet anzupassen. Sofern dies mit der Systemspezifikation vereinbar ist, kann der Test weiter ausgeführt werden und ein möglicherweise fälschliches Testurteil wird somit vermieden.

Der Test einer Implementation verringert die Fehlerquote wesentlich, das System wird nichtsdestotrotz nur hinsichtlich seiner Spezifikation überprüft. In vielen Fällen genügt bereits diese Spezifikation den Anforderungen des Auftraggebers nicht oder nicht vollständig. Um das Fehlerrisiko weiter zu reduzieren, müssen also auch die Modelle der Software selbst verifiziert werden. Dies geschieht in einer Modellverifikationsphase, die im Softwareentwicklungsprozess noch vor dem Testen stattfindet. Auch hier muss wieder die Explosion des Zustandsraumes durch eine geeignete Modellabstraktion verhindert werden.

Eine Folge von Modellabstraktionen im Zusammenhang mit der Modellverifikation sind sogenannte *False Negatives*, Gegenbeispiele, die einen Fehler im abstrahierten Modell anzeigen, jedoch im originalen Modell nicht nachzuvollziehen sind. Gewöhnlich werden diese False Negatives ignoriert, allerdings wird in dieser Dissertation auch eine Methode entwickelt, die dieses Wissen um mögliche Fehler nutzbar macht. Dafür wird das gewonnene Gegenbeispiel weiter abstrahiert und von ihm ein Fehlermuster abgeleitet. Mit Hilfe eines Constraint-Solvers wird dann nach einem tatsächlichen Gegenbeispiel im System gesucht.

Curriculum Vitæ

Jens R. Calamé was born on June 21st, 1979 in Hamburg, Germany. In 1999, he graduated from the Johann-Rist-Gymnasium in Wedel, Germany, and started studying at the Institute for Computer Science at the University of Potsdam, Germany, in October that year. In 2004, he received his university diploma (Diplom-Informatiker) with a thesis on the test of software agents and inter-agent-connectors under the supervision of Prof. Dr. E. Horn and Prof. Dr. I. Schieferdecker.

In July 2004, Calamé started his PhD study in the Software Engineering Group 2 (Specification and Analysis of Embedded Systems) at the Centrum Wiskunde & Informatica in Amsterdam, The Netherlands. Under the supervision of Prof. Dr. Jaco van de Pol and Prof. Dr. Wan Fokkink, he worked on the improvement of software test generation and execution techniques for reactive systems with data, as well as on the improvement of debugging techniques for model checking. His work was carried out under the auspices of the TT-Medal project (Tests & Testing Methodologies for Advanced Languages) and the BSIK/BRICKS project (Basic Research in Informatics for Creating the Knowledge Society). The present thesis contains the results of this work.

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*.

Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*.

Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source*

Code. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20