Valencia, December 2017

Rewriting Logic Techniques for Program Analysis and Optimization

Author
Julia Sapiña Sanchis

Supervisors
María Alpuente Frasnedo
Demis Ballis

Universitat Politècnica de València

A dissertation submitted in fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Rewriting Logic Techniques for Program Analysis and Optimization

by

Julia Sapiña Sanchis

This thesis has been partially supported by the FPI-UPV grant SP2013-0083 and UPV mobility grant VIIT-3946.

To my mother.

Abstract

This thesis proposes a dynamic analysis methodology for improving the diagnosis of erroneous Maude programs. The key idea is to combine runtime assertion checking and dynamic trace slicing for automatically catching errors at runtime while reducing the size and complexity of the erroneous traces to be analyzed (i.e., those leading to states that fail to satisfy the assertions). In the event of an assertion violation, the slicing criterion is automatically inferred, which facilitates the user to rapidly pinpoint the source of the error.

First, a technique is formalized that aims at automatically detecting anomalous deviations of the intended program behavior (error symptoms) by using assertions that are checked at runtime. This technique supports two types of user-defined assertions: functional assertions (which constrain deterministic function calls) and system assertions (which specify system state invariants). The proposed dynamic checking is provably sound in the sense that all errors flagged definitely signal a violation of the specifications. Then, upon eventual assertion violations, accurate trace slices (i.e., simplified yet precise execution traces) are generated automatically, which help identify the cause of the error. Moreover, the technique also suggests a possible repair for the rules involved in the generation of the erroneous states.

The proposed methodology is based on (i) a logical notation for specifying assertions that are imposed on execution runs; (ii) a runtime checking technique that dynamically tests the assertions; and (iii) a mechanism based on (equational) least general generalization that automatically derives accurate criteria for slicing from falsified assertions.

Finally, an implementation of the proposed technique is presented in the assertion-based, dynamic analyzer ABETS, which shows how the forward and backward tracking of asserted program properties leads to a thorough trace analysis algorithm that can be used for program diagnosis and debugging.

Resumen

Esta tesis propone una metodología de análisis dinámico que mejora el diagnóstico de programas erróneos escritos en el lenguaje Maude. La idea clave es combinar técnicas de verificación de aserciones en tiempo de ejecución con la fragmentación dinámica de trazas de ejecución para detectar automáticamente errores en tiempo de ejecución, al tiempo que se reduce el tamaño y la complejidad de las trazas a analizar. En el caso de violarse una aserción, se infiere automáticamente el criterio de fragmentación, lo que facilita al usuario identificar rápidamente la fuente del error.

En primer lugar, la tesis formaliza una técnica destinada a detectar automáticamente eventuales desviaciones del comportamiento deseado del programa (síntomas de error). Esta técnica soporta dos tipos de aserciones definidas por el usuario: aserciones funcionales (que restringen llamadas a funciones deterministas) y aserciones de sistema (que especifican los invariantes de estado del sistema). La técnica de verificación dinámica propuesta es demostrablemente correcta en el sentido de que todos los errores señalados definitivamente delatan la violación de las aserciones. Tras eventuales violaciones de aserciones, se generan automáticamente trazas fragmentadas (es decir, trazas simplificadas pero igualmente precisas) que ayudan a identificar la causa del error. Además, la técnica también sugiere una posible reparación para las reglas implicadas en la generación de los estados erróneos.

La metodología propuesta se basa en (i) una notación lógica para especificar las aserciones que se imponen a la ejecución; (ii) una técnica de verificación aplicable en tiempo de ejecución que comprueba dinámicamente las aserciones; y (iii) un mecanismo basado en la generalización (ecuacional) menos general que automáticamente obtiene criterios precisos para fragmentar trazas de ejecución a partir de aserciones falsificadas.

Por último, se presenta una implementación de la técnica propuesta en la herramienta de análisis dinámico basado en aserciones ABETS, que muestra cómo es posible combinar el trazado de las propiedades asertadas del programa para obtener un algoritmo preciso de análisis de trazas que resulta útil para el diagnóstico y la depuración de programas.

Resum

Esta tesi proposa una metodologia d'anàlisi dinàmica que millora el diagnòstic de programes erronis escrits en el llenguatge Maude. La idea clau és combinar tècniques de verificació d'assercions en temps d'execució amb la fragmentació dinàmica de traces d'execució per a detectar automàticament errors en temps d'execució, alhora que es reduïx la grandària i la complexitat de les traces a analitzar. En el cas de violar-se una asserció, s'inferix automàticament el criteri de fragmentació, la qual cosa facilita a l'usuari identificar ràpidament la font de l'error.

En primer lloc, la tesi formalitza una tècnica destinada a detectar automàticament eventuals desviacions del comportament desitjat del programa (símptomes d'error). Esta tècnica suporta dos tipus d'assercions definides per l'usuari: assercions funcionals (que restringixen crides a funcions deterministes) i assercions de sistema (que especifiquen els invariants d'estat del sistema). La tècnica de verificació dinàmica proposta és demostrablement correcta en el sentit que tots els errors assenyalats definitivament delaten la violació de les assercions. Davant eventuals violacions d'assercions, es generen automàticament traces fragmentades (és a dir, traces simplificades però igualment precises) que ajuden a identificar la causa de l'error. A més, la tècnica també suggerix una possible reparació de les regles implicades en la generació dels estats erronis.

La metodologia proposada es basa en (i) una notació lògica per a especificar les assercions que s'imposen a l'execució; (ii) una tècnica de verificació aplicable en temps d'execució que comprova dinàmicament les assercions; i (iii) un mecanisme basat en la generalització (ecuacional) menys general que automàticament obté criteris precisos per a fragmentar traces d'execució a partir d'assercions falsificades.

Finalment, es presenta una implementació de la tècnica proposta en la ferramenta d'anàlisi dinàmica basat en assercions ABETS, que mostra com és possible combinar el traçat cap avant i cap arrere de les propietats assertades del programa per a obtindre un algoritme precís d'anàlisi de traces que resulta útil per al diagnòstic i la depuració de programes.

Acknowledgments

I would like to dedicate a few words to the people without whom this thesis would never have been possible.

My first and deepest gratitude is to my supervisor, María Alpuente, for giving me the opportunity to join the Extensions of Logic Programming (ELP) research group. María not only has an impeccable work ethic, but also inspires the people around her with her brilliant mind. Rarely found these days, María combines her outstanding professional facet with an even more remarkable personal one. Under her supervision, instructions, and advice, I have learned how to be a researcher, and, more importantly, I have grown as a person and hopefully become a better one. Over the past weeks I have been thinking about how to write these lines without falling into *clichés* that weaken my words. My conclusion is that the most genuine and sincere compliment I can dedicate to her is that, because of all the good qualities and even small gestures that she shares with her, María has always reminded me of my mother, whom I deeply admire for her intelligence, courage, kindness, and ability to sacrifice for others.

I want to extend this gratitude to my other supervisor, Demis Ballis, whose enthusiasm and guidance have greatly helped me overcome the low moments of this thesis; I really enjoy working with you, my friend. Also to my colleague, Francisco Frechina, who co-supervised me during my Master's degree; I miss our inspiring discussions about our research and how we managed to solve the problems that the other one said could not be solved just to be right.

Thanks also go to my senior colleagues of the ELP. To Santiago Escobar for being so authentic; whenever I have to travel, I still remember how you made us all laugh on the way to my first conference, especially on the way back. To Alicia Villanueva and Raúl Gutiérrez for many, many things, but especially, for all those brief chats and technical discussions at the coffee machine after lunch that I very much enjoy every day. To Salvador Lucas for *terminating* with such dedication all the theoretical doubts I had every time I consulted you. And last but not least, thanks to the members of the Data Mining, Machine Intelligence and Inductive Programming (DMIP) team, Mª José Ramírez, Cèsar Ferri, and Jose Hernández-Orallo for their selfless support and help with anything I needed.

Thanks to such a wonderful group of fellows with whom I have shared my workspace these past years. To Fernando Martínez-Plumed, for his priceless advice and conversations about nothing and everything and for being the only *constant* in a lab full of *variables*. Thanks to Sonia Santiago, for having such a good heart and will to always help others; we really miss you here. To Lidia Contreras-Ochando, for bringing the joy back to the lab after so many colleagues finished their theses and left. To Javier Espert, our former server admin, for helping me *install* myself when I first arrived. To Javier Insa and Marco Feliú for the brief, yet pleasant, moments we shared together. To Daniel Pardo for being so quiet so that the rest of us could make so much noise. And to David Nieves, our last addition, for the good things that are surely yet to come.

Thanks to our *brothers* of the Multi-paradigm Software Technology (MiST) research group, Germán Vidal, Marisa Llorens, Carlos Herrero, Javier Oliver, Josep Silva, Salvador Tamarit, David Insa, Adrián Palacios, and Sergio Pérez because, although we have not coincided much professionally, I always knew I could count on you whenever I needed it.

Finally, I want to especially thank Moreno Falaschi and his family for making my stay in Siena feel like home, Eva Onaindia for her hard work coordinating the new doctoral program and for solving any doubts or problems I had (sometimes even before knowing I had them), and Alberto Lluch Lafuente, Francesco Tiezzi, and Alberto Verdejo for their valuable feedback as reviewers of this thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Assertion checking is one of the most useful available techniques for detecting program faults. Although not universally used, assertions seem to have widely infiltrated into common programming practice, primarily for finding bugs in the later stages of development. In dynamic assertion checking, assertions are program properties that are traditionally used to express conditions that should hold at runtime. By finding inconsistencies between specified properties and the program code, dynamic assertion checking can prove that the code is incorrect. Moreover, since an assertion failure usually reports an error, the user can direct her attention to the location at which the logical inconsistency is detected and trace it back to the sources that are responsible for the erroneous behavior.

Program slicing [Weiser, 1981, Korel and Laski, 1988, Korel and Rilling, 1997] is another well-established technique in software engineering with increasing recognition for error diagnosis and program comprehension, since it allows developers to focus on the code fragment that is relevant to the piece of information that she wants to observe. Program slicing is a program transformation method that also finds applications in program optimization and useless-code elimination. The basic idea of program slicing is to isolate a subset of statements that either contribute to the values of a set of variables at a given point, or that are influenced by the values of a given set of variables. The first approach corresponds to forms of backward slicing, whereas the second one corresponds to forward slicing.

While both assertion-checking and program slicing are classically used in program debugging and understanding, their benefits have been poorly exploited within the Maude community to date. This also applies to program repair methods that help correct or simplify a program by eliminating either useless, unimportant, or erroneous parts of the program or of the program state space.

This chapter introduces the main motivation of the thesis in Section 1.1 and summarizes the thesis contribution in Section 1.2. An outline of the dissertation is presented in Section 1.3. Finally, Section 1.4 summarizes the list of publications and software tools developed by the author as part of her thesis.

1.1 Motivation

Program debugging is crucial for reliable software development because the size and complexity of modern software systems make (requirement and design) specifications extremely difficult and error-prone. Unfortunately, debugging is generally a burdensome process that involves a large portion of the software development effort in order to locate the actual cause of observable misbehaviors. In order to mitigate the costs of debugging, automated tools and techniques are required to help identify the root cause of errors. This thesis proposes a general approach for improving the debugging of Maude programs that is based on systematically combining runtime assertion checking and trace (and program) fragmentation.

Maude [Clavel et al., 2007] is a high-performance language and system that provides a powerful variety of correctness tools and techniques including rapid prototyping, state space exploration, and model checking of temporal formulas. Maude programs correspond to specifications in rewriting logic [Meseguer, 1992], which is an extension of membership equational logic [Meseguer, 1998] that, besides supporting equations and allowing the elements of a type or *sort* to be characterized by means of membership axioms, adds rewrite rules that can be non-deterministic and represent state transitions in a concurrent system. Thanks to its reflective design and meta-level capabilities, the Maude system provides powerful and highly efficient metaprogramming facilities. This has further contributed to its success, giving support to the development of sophisticated tools and techniques for the modeling and analysis of Maude programs, such as LTLR model checking [Bae and Meseguer, 2015], abstract certification [Alba-Castro et al., 2010], Web verification [Alpuente et al., 2009a, Alpuente et al., 2010b], narrowing-based code-carrying theory [Alpuente et al., 2010a], *etc.* (for a survey of the related literature, see [Martí-Oliet et al., 2012]). The use of slicing for debugging Maude programs is discussed in [Alpuente et al., 2014a], and it relies on a rich and highly dynamic parameterized scheme for exploring rewriting logic computations defined in [Alpuente et al., 2014b, Alpuente et al., 2015a] that can significantly reduce the size and complexity of the runs under examination by automatically slicing both programs and computation traces. However, Maude does not currently provide general support for asserting properties that are dynamically-checked. The main aim of this work is to fill this gap by providing Maude with runtime assertion-checking capabilities. This is done by first introducing a simple assertion language that suffices for the purpose of improving error diagnosis and debugging in the context of rewriting logic. This thesis follows the approach of modern specification and verification systems such as Spec# [Barnett et al., 2004] or the Java Modeling Language [Burdy et al., 2003], where the specification language is typically an extension of the underlying programming language and specifications are used as *contracts* that guarantee certain properties to hold at a number of execution states (e.g., before or after a given function call [Leavens and Cheon, 2005]). This approach is of practical interest because it facilitates the job of programmers.

This thesis presents a dynamic analysis framework that seamlessly combines runtime assertion checking, trace slicing, and automated program repair for improving the diagnosis of Maude programs.

1.2 Contributions of the Thesis

The main contributions of the thesis can be summarized as follows:

1. A simple yet powerful logic assertion language, which allows Maude developers to easily express properties that system states and computations must obey.

2. A logic semantics for the assertion language, which formalizes the notion of satisfaction of an assertional specification that contains system assertions (which specify system state invariants) and functional assertions (which constrain the result of deterministic function calls).

3. A runtime verification technique that automatically checks Maude programs against assertional specifications, which can be trivially adapted to perform offline verification i.e., verification of previously deployed execution traces.

4. A formal technique for automatically computing accurate error symptoms, i.e., the pieces of information that are directly responsible for the violation of an assertion, together with an improvement of the technique

that produces more refined error symptoms by identifying (and thus ignoring) the subformulas of the assertion that play no role in its violation.

5. A methodology that combines both runtime verification and trace slicing by automatically inferring accurate slicing criteria from previously detected faulty execution traces.

6. A novel automated technique that suggests suitable repairs for the rules that are involved in the generation of erroneous system states (i.e., those states that cause the violation of a system assertion).

7. An implementation of all the above techniques in the ABETS system, which helps analyze Maude as well as Full Maude [Clavel et al., 2007] programs, together with an experimental evaluation of all implemented techniques.

1.3 Plan of the Thesis

The remainder of the thesis is structured as follows:

- **Chapter 2: Preliminaries.** This chapter briefly introduces some basic knowledge on term rewriting and rewriting logic that is fundamental for this dissertation. The chapter also introduces the specific notation that will be used throughout the manuscript.

- **Chapter 3: The Maude System and Language.** This chapter provides a summary of the most relevant features of the high-performance declarative language and system Maude. Non-experienced readers who are not familiarized with Maude may find it very useful to understand the examples developed in the thesis.

- **Chapter 4: Inspection of Rewriting Logic Computations.** This chapter summarizes the rewriting-based, trace instrumentation and computation exploration techniques of [Alpuente et al., 2013b, Alpuente et al., 2015a], which are at the core of the analysis and inspection techniques proposed in the thesis. The chapter also introduces the two running examples (i.e., a model of a Stock Exchange system written in Core Maude and an object-oriented, distributed online car-rental store written in Full Maude) that are used to better illustrate the proposed techniques and methodologies.

- **Chapter 5: Runtime Verification of Maude Programs.** This chapter first introduces a simple assertion language for Maude programs that allows developers to express properties that are both *executable* in user-defined programs and quite versatile. The assertion language distinguishes two types of assertions: (i) functional assertions, for specifying properties of functions defined by an equational theory; and (ii) system assertions, which allow properties concerning the system's execution to be expressed. Assertions are provided with a semantics based on a notion of logic satisfaction that formally establishes what means for an equational simplification trace (resp. a system state) to satisfy functional (resp. system) assertions. The concept of *error symptoms* (i.e., the pieces of information that are directly related to an assertion violation) is then formalized. The chapter finishes by formulating a verification technique that dinamically checks assertions at runtime.

- **Chapter 6: Automated Debugging of Maude Programs.** This chapter presents an assertion-based trace slicing and verification technique for debugging rewriting logic computations, which combines both trace slicing and the dynamic verification technique of Chapter 5. This technique exploits the information that is dynamically computed when an assertion fails (i.e., the *error symptoms*) to help correlate the simple external evidence of the error with the complexity of searching possible program locations for the faults that caused the error. A refinement of the basic technique for automatically inferring the slicing criteria is also formalized. Finally, the chapter formulates a new program repair technique that automatically suggests program corrections to fix those program faults that are signaled by the violation of a system assertion.

- **Chapter 7: The ABETS System.** This chapter describes an implementation of the proposed techniques in the assertion-based, dynamic trace analyzer ABETS. The chapter explains the functionality and main features of ABETS and demonstrates its *bug-catching* capabilities by reproducing a detailed debugging session. Moreover, an in-depth experimental evaluation of the system is also provided, which assesses the practicality of the tool.

- **Chapter 8: Mau-Dev: A Developer Extension of Maude.** This chapter explains those novel implementation details and optimizations that have boosted the performance of the ABETS system. This has been achieved by reimplementing the functions that are most frequently used in ABETS as new, highly efficient (Mau-Dev) built-in metalevel operations. At the end of this chapter an experimental evaluation is performed that highlights the performance and effectiveness of this system extension.

- **Chapter 9: Conclusion.** This final chapter provides an overview of the related literature and concludes by discussing further investigations that are planned as future work.

1.4 Bibliographic Remarks

Let us conclude this introduction with some bibliographic remarks:

- The exploration technique discussed in Chapter 4 was originally presented in [Alpuente et al., 2013b, Alpuente et al., 2014b, Alpuente et al., 2015a] and included in [Frechina, 2014]. Chapter 4 briefly summarizes it in order to provide the reader wih the necessary background knowledge that is required to fully understand this dissertation.

- The runtime assertion-based slicing and debugging methodologies that are formalized in Chapters 5 and 6 were originally presented in [Alpuente et al., 2015b] and [Alpuente et al., 2016a, Alpuente et al., 2016b] respectively.

- The ABETS tool, which is described in Chapter 7, was originally published in [Alpuente et al., 2016b] together with detailed benchmark experiments.

- Progressive additions to the Mau-Dev system, which is developed in Chapter 8, were presented in [Alpuente et al., 2015a], [Alpuente et al., 2016b], and [Alpuente et al., 2017].

- Finally, the ANIMA and ABETS tools described in Chapters 4 and 7 inherited some key features originally developed in the M.Sc. thesis [Sapiña, 2013] for the predecessor tool *i*Julienne [Alpuente et al., 2013a]. The tools were then significantly improved by developing important optimizations that were subsequently ported back to *i*Julienne.

The following subsections summarize the full list of publications and tools.

1.4.1 List of Publications

The work that has led to the development of this Ph.D. thesis has been published in the following journals and conference proceedings.

[Alpuente et al., 2013a] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. **Slicing-Based Trace Analysis of Rewriting Logic Specifications with iJulienne**. In Proceedings of the 22nd European Symposium on Programming, volume 7792 of Lecture Notes in Computer Science, pages 121-124. Springer, 2013.

[Alpuente et al., 2013b] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. **Parametric Exploration of Rewriting Logic Computations**. In Proceedings of the 5th International Symposium on Symbolic Computation in Software Science, volume 15 of EasyChair Proceedings in Computing, pages 4-18. EasyChair, 2013.

[Alpuente et al., 2014b] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. **Inspecting Rewriting Logic Computations (in a Parametric and Stepwise Way)**. In Proceedings of Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi, volume 8373 of Lecture Notes in Computer Science, pages 229-255. Springer, 2014.

[Alpuente et al., 2015b] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. **Combining Runtime Checking and Slicing to Improve Maude Error Diagnosis**. In Proceedings of Logic, Rewriting, and Concurrency: Essays Dedicated to José Meseguer on the Occasion of his 65th Birthday (LRC 2015), volume 9200 of Lecture Notes in Computer Science, pages 72-96. Springer, 2015.

[Alpuente et al., 2015a] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. **Exploring Conditional Rewriting Logic Computations**. In Journal of Symbolic Computation, volume 69, pages 3-39. Elsevier, 2015.

[Alpuente et al., 2016a] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. **Debugging Maude Programs via Runtime Assertion Checking and Trace Slicing**. In Journal of Logical and Algebraic Methods in Programming, volume 85, issue 5, pages 707-736. Elsevier, 2016.

[Alpuente et al., 2016b] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. **Assertion-based Analysis via Slicing with ABETS**. In Theory and

Practice of Logic Programming, volume 16, issue 5-6, pages 515-532. Cambridge University Press, 2016.

[Alpuente et al., 2017] M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Sapiña. **Inspecting Maude Variants with GLINTS**. In Theory and Practice of Logic Programming, volume 17, issue 5-6, pages 689-707. Cambridge University Press, 2017.

1.4.2 List of Tools

During the development of the thesis the following software systems have been implemented.

1. **ANIMA**, an interactive program analyzer for Maude available at `http://safe-tools.dsic.upv.es/anima`.
2. **ABETS**, an assertion-based trace slicer and checker available at `http://safe-tools.dsic.upv.es/abets`.
3. **Mau-Dev**, a developer extension of Maude available at `http://safe-tools.dsic.upv.es/maudev`.

Chapter 2

Preliminaries

Rewriting logic [Meseguer, 1992] is a powerful yet simple computational logic that can express both concurrent computation (by means of rewrite rules) and logical deduction (by means of equations) that is particularly suitable for dealing with highly non-deterministic concurrent systems and computations. Rewriting logic is also a sound and complete semantic framework in which many different models of concurrency, distributed algorithms, programming languages, and software and hardware modeling languages can be naturally represented, executed, and analyzed as rewrite theories [Meseguer, 2012].

This chapter recalls some important notions about rewriting logic that are relevant to this thesis. We assume some basic knowledge of term rewriting [TeReSe, 2003] and rewriting logic [Meseguer, 1992]. For deeper details, we refer to [Klop, 1992, Baader and Nipkow, 1998, Ohlebusch, 2002, TeReSe, 2003, Meseguer, 1992]. The chapter is structured as follows. Section 2.1 introduces the building blocks of rewriting logic. Section 2.2 details the sophisticated rewriting mechanism of rewriting logic. Finally, Section 2.3 explains generalization modulo equational theories, which is the dual operation of equational unification and is a key component of the diagnosis and repair methodologies that are proposed in this dissertation.

2.1 Basic Concepts of Rewriting Logic

This section introduces some basic concepts of the rewriting logic framework that are fundamental for the thesis.

2.1.1 The Term Language of Rewriting Logic

Let Σ be an order-sorted *signature* with a finite poset of sorts $(S, <)$ that models the usual subsort relation [Clavel et al., 2016]. The connected components of $(S, <)$ are the equivalence classes $[s]$ corresponding to the least equivalence relation $\equiv_<$ containing $<$. A signature Σ allows operators to be specified together with their type structure by means of suitable sets of sorts, subsorts, and kinds. The kinds allow equivalent sorts[1] to be grouped together and, intuitively, can be considered as an *error supersort*. Therefore, terms (built over Σ) that have a kind but not a sort are understood to be undefined or error terms. $\mathcal{T}(\Sigma, \mathcal{V})_s$ and $\mathcal{T}(\Sigma)_s$ are the sets of terms and ground terms of sort s, respectively. Given a term t, by $\mathcal{V}ar(t)$, we denote the set of variables that occur in t. Given a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ and a sort $s \in \Sigma$, then by $t : s$ we denote that t is of sort s. We write $\mathcal{T}(\Sigma, \mathcal{V})$ and $\mathcal{T}(\Sigma)$ for the corresponding term algebras.

A *position* w in a term t is represented by a sequence of natural numbers that addresses a subterm of t (Λ denotes the empty sequence, i.e., the root position). By notation $w_1.w_2$, we denote the concatenation of positions (sequences) w_1 and w_2. Given a term t, we let $\mathcal{P}os(t)$ denote the set of positions of t. Given a position w of a term t, $\mathcal{P}os_w(t) = \{w.w' \mid w.w' \in \mathcal{P}os(t)\}$. By $\mathcal{VP}os(t)$, we denote the set of variable positions of a term t. Positions are ordered by the prefix ordering, that is, given the positions w_1 and w_2, $w_1 \leq w_2$ if there exists a position u such that $w_1.u = w_2$. Given two positions w_1 and w_2, we say that w_1 and w_2 are not comparable iff $w_1 \not\leq w_2$ and $w_2 \not\leq w_1$. By $t|_w$, we denote the *subterm* of t at position w, and by $t[s]_w$, we denote the result of replacing the subterm $t|_w$ by the term s in t.

Given a binary relation $\rightsquigarrow$, we define the usual *transitive* (resp., *transitive and reflexive*) closure of $\rightsquigarrow$ by $\rightsquigarrow^+$ (resp., $\rightsquigarrow^*$).

A *substitution* $\sigma \equiv \{X_1/t_1, X_2/t_2, \ldots, X_n/t_n\}$ is a mapping from the set of variables $\mathcal{V}$ to the set of terms $\mathcal{T}(\Sigma, \mathcal{V})$, which is equal to the identity except for a finite set of variables $\{X_1, \ldots, X_n\}$. The *domain* of σ is the set $\mathcal{D}om(\sigma) = \{X \in \mathcal{V} \mid X\sigma \neq X\}$. By $\{\}$, we denote the *identity* substitution. The *application of a substitution* σ to a term t, denoted $t\sigma$, is defined by induction on the structure of terms:

$$t\sigma = \begin{cases} X\sigma & \text{if } t = X, X \in \mathcal{V} \\ f(t_1\sigma, \ldots, t_n\sigma) & \text{if } t = f(t_1, \ldots, t_n), n \geq 0 \end{cases}$$

Given two terms s and t, a substitution σ is a *matcher* of t in s, if $s\sigma = t$. The term t is an *instance* of the term s, iff there exists a matcher σ of t in s. We also say that s is *more general* than t (and that t is an *instance* of s) iff there

[1]Two sorts are in the same equivalence class if and only if they belong to the same connected component. Sorts are user-defined and explicitly declared in Σ, while kinds are implicitly associated with equivalence classes of sorts.

exists a matcher σ such that $s\sigma = t$. Substitutions can also be represented as sets of equations. More formally, given a substitution $\sigma = \{X_1/t_1, ..., X_n/t_n\}$, its equational representation is $\widehat{\sigma} = \{X_1 = t_1, ..., X_n = t_n\}$.

Given two substitutions σ and θ, their composition is written $\sigma\theta$. Note that applying $\sigma\theta$ to a term t is equivalent to first compute $t\sigma$ and then apply θ to the result.

Given two substitutions σ and θ, their parallel composition (or reconciliation) [Palamidessi, 1990], is written $\sigma \Uparrow \theta$ and amounts to compute the most general unifier (mgu, see [Martelli and Montanari, 1982]) of the set resulting from the union of the equational representations of both σ and θ (i.e., $\sigma \Uparrow \theta = mgu(\widehat{\sigma} \cup \widehat{\theta})$).

2.1.2 Conditional Equations, Rewrite Rules, and Membership Axioms

A *conditional equation* is an expression of the form $[l] : \lambda = \rho$ *if* C, where $\lambda, \rho \in \mathcal{T}(\Sigma, \mathcal{V})$, $\mathcal{V}ar(\rho) \subseteq \mathcal{V}ar(\lambda)$, C is a condition that must be fulfilled, and l is a label, i.e., a name that identifies the equation. A *condition* C is an expression of the form $c_1 \wedge ... \wedge c_n$, with $n \geq 0$, where each c_i is either an equation (which requires the equality of two terms), a membership, or a rewrite expression (which requires the existence of a rewriting chain between the terms).

A *conditional rewrite rule* is an expression of the form $[l] : \lambda \Rightarrow \rho$ *if* C, where $\lambda, \rho \in \mathcal{T}(\Sigma, \mathcal{V})$, $\mathcal{V}ar(\rho) \subseteq \mathcal{V}ar(\lambda)$, C is a condition that must be fulfilled, and l is a label.

A *conditional membership axiom* is an expression of the form $[l]$: $\lambda : s$ *if* C, where λ is a term, s is a sort, C is a condition that must be fulfilled, and l is a label. Membership axioms assert that terms that match λ modulo equational axioms such as associativity, commutativity, and unity, have a specific sort s.

When no confusion can arise, rule, equation, and membership axioms labels are often omitted. Also, for the unconditional case where the condition is empty, equations (resp. rules, resp. memberships) adopt the simplified notation $\lambda = \rho$ (resp. $\lambda \Rightarrow \rho$, resp. $\lambda : s$). The term λ (resp., ρ) is called *left-hand side* (resp. *right-hand side*) of the rule $\lambda \Rightarrow \rho$ (resp. equation $\lambda = \rho$).

2.1.3 Conditional Rewrite Theories

A *membership equational theory* is a pair (Σ, E), where Σ is an order-sorted signature, $E = \Delta \cup B$, with Δ a collection of conditional (oriented) equations and membership axioms), and B a collection of equational axioms (i.e., algebraic laws such as associativity, commutativity, and unity) that can be associated with any binary operator of Σ. The equational theory (Σ, E) induces a congruence relation on the term algebra $\mathcal{T}(\Sigma, \mathcal{V})$, which is denoted by $=_E$.

The static state structure as well as the dynamic behavior of a concurrent system can be formalized as a rewriting logic specification that encodes a *conditional rewrite theory*. More specifically, a *conditional rewrite theory* (or simply *rewrite theory*) is a triple $\mathcal{R} = (\Sigma, E, R)$, where:

(i) (Σ, E) is a membership equational theory that contains the system data types to be defined via equations, equational axioms, and membership axioms.

(ii) R is a set of conditional rewrite rules.

Intuitively, (Σ, E) allows system states to be formalized as terms, while rules in R specify general patterns that are used to model state transitions and allow the dynamics of the system to be specified. More specifically, the system evolves by applying the rules of the rewrite theory R to the system states by means of *rewriting modulo* E. An in-depth explanation of the operational semantics underlying rewriting logic is summarized in the following section.

2.2 Rewriting in Rewriting Logic

Let us consider a conditional rewrite theory (Σ, E, R), with $E = \Delta \cup B$, where Δ is a set of conditional equations and membership axioms, and B is a set of equational axioms associated with some binary operators in Σ. The conditional rewriting modulo E relation (in symbols, $\rightarrow_{R/E}$) can be defined by lifting the usual conditional rewrite relation on terms [Klop, 1992] to the E-congruence classes $[t]_E$ on the term algebra $\mathcal{T}(\Sigma, \mathcal{V})$ that are induced by $=_E$ [Bruni and Meseguer, 2006]. In other words, $[t]_E$ is the class of all terms that are equal to t *modulo* E. Unfortunately, $\rightarrow_{R/E}$ is, in general, undecidable since a rewrite step $t \rightarrow_{R/E} t'$ involves searching through the possibly infinite equivalence classes $[t]_E$ and $[t']_E$.

For a conditional rewrite theory to be executable, its equations Δ should be Church-Rosser [Church and Rosser, 1936, Meseguer, 1992] and terminating [Turing, 1937, Dershowitz, 1987] modulo the given axioms B, and their rules R should be (ground) coherent [Viry, 2002, Durán and Meseguer, 2012] with

Δ modulo B. This allows developers to implement conditional rewriting $\rightarrow_{R/E}$ with R modulo E by means of two much simpler relations, namely $\rightarrow_{\Delta,B}$ and $\rightarrow_{R,B}$, which allow rules, equations and memberships to be intermixed in the rewriting process by simply using an algorithm of matching modulo B.

The relation $\rightarrow_{R\cup\Delta,B}$ is defined as $\rightarrow_{R,B} \cup \rightarrow_{\Delta,B}$. Roughly speaking, the relation $\rightarrow_{\Delta,B}$ uses the equations of Δ (oriented from left to right) as simplification rules. Thus, by repeatedly applying the equations as simplification rules from a given term t, we eventually reach a term $t\downarrow_{\Delta,B}$ to which no further equations can be applied. The term $t\downarrow_{\Delta,B}$ is called a *canonical (or normal)* form of t with respect to Δ modulo B. An *equational simplification* of a term t in Δ modulo B is a rewrite sequence of the form $t \rightarrow^{*}_{\Delta,B} t\downarrow_{\Delta,B}$. Informally, the relation $\rightarrow_{R,B}$ implements rewriting with the rules of R, which might be non-terminating and non-confluent, whereas Δ is required to be Church-Rosser and terminating modulo B in order to guarantee the existence and unicity (modulo B) of a canonical form with respect to Δ for any term [Clavel et al., 2016].

Terms are rewritten into canonical forms according to their sort structure, which is induced by the signature Σ and the membership axioms specified in Δ. In particular, through conditional membership axioms of the form $\lambda : s$ *if* C, we can assert that any term B-matching λ has a specific sort s whenever a condition C holds. Equational simplification of terms is naturally lifted to substitutions as follows: given $\sigma = \{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$, we define the *normalized* substitution $\sigma\downarrow_{\Delta,B} = \{x_i/(t_i\downarrow_{\Delta,B})\}_{i=1}^{n}$.

Formally, $\rightarrow_{R,B}$ and $\rightarrow_{\Delta,B}$ are defined as follows. Given a conditional rewrite rule of the form $[l] : \lambda \Rightarrow \rho$ *if* $C \in R$ (resp., an equation $[l] : \lambda = \rho$ *if* $C \in \Delta$), a substitution σ, a term t, and a position w of t, $t \overset{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp., $t \overset{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$) iff $\lambda\sigma =_B t|_w$, $t' = t[\rho\sigma]_w$, and $C\sigma$ *evaluates to true*. When no confusion arises, we simply write $t \rightarrow_{R,B} t'$ (resp. $t \rightarrow_{\Delta,B} t'$) instead of $t \overset{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp. $t \overset{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$). Roughly speaking, a conditional rewrite step on the term t applies a rewrite rule/equation to t by replacing a *red*ucible (sub-)*ex*pression of t (namely $t|_w$), called the *redex*, by its contracted version $\rho\sigma$, called the *contractum*, whenever the condition $C\sigma$ is fulfilled. Note that the evaluation of a condition C is typically a recursive process since it may involve further (conditional) rewrites in order to normalize C to true.

Under appropriate coherence conditions [Durán and Meseguer, 2012] on the rewrite theory, a rewrite step $s \rightarrow_{R/E} t$ modulo E on a term s can be implemented without loss of completeness by applying a rewrite strategy that involves the repeated application of the two following basic steps [Durán and Meseguer, 2012]:

1. **Equational simplification of** s **in** Δ **modulo** B, that is, reduce s using $\rightarrow_{\Delta,B}$ until the canonical form with respect to Δ modulo B $(s\downarrow_{\Delta,B})$ is reached;

2. **Rewrite** $(s\downarrow_{\Delta,B})$ **in** R **modulo** B to t' using $\rightarrow_{R,B}$, where $t' \in [t]_E$.

Note that, for the one-step rewrite relation $\rightarrow_{R/E}$ to be well-defined, in the remainder of this thesis we assume that the evaluation of all conditions for all conditional rules in R terminates.
An *execution trace* (or *computation*) $\mathcal{C}$ for s_0 in the conditional rewrite theory $(\Sigma, \Delta \cup B, R)$ is then deployed as the (possibly infinite) rewrite sequence

$$s_0 \rightarrow^*_{\Delta,B} s_0{\downarrow_{\Delta,B}} \rightarrow_{R,B} s_1 \rightarrow^*_{\Delta,B} s_1{\downarrow_{\Delta,B}} \rightarrow_{R,B} \ldots$$

that interleaves $\rightarrow_{\Delta,B}$ rewrite steps and $\rightarrow_{R,B}$ rewrite steps following the strategy mentioned above. After each conditional rewrite step using $\rightarrow_{R,B}$, in general, the resulting term s_i, $i = 1, \ldots, n$, is not in canonical form. Therefore, it is normalized before the subsequent rewrite step with $\rightarrow_{R,B}$ is performed. Also, in the precise strategy adopted by Maude, the last term of a finite computation is finally normalized before the result is delivered. By ε, we denote the *empty* computation. Therefore, any computation can be interpreted as a sequence of juxtaposed $\rightarrow_{R,B}$ and $\rightarrow^*_{\Delta,B}$ transitions, with an additional equational simplification $\rightarrow^*_{\Delta,B}$ (if needed) at the beginning of the computation as depicted below.

$$\overbrace{s_0 \rightarrow^*_{\Delta,B} s_0{\downarrow_{\Delta,B}} \rightarrow_{R,B} s_1 \rightarrow^*_{\Delta,B} s_1{\downarrow_{\Delta,B}}}\underbrace{\phantom{s_1{\downarrow_{\Delta,B}}} \rightarrow_{R,B} s_2 \rightarrow^*_{\Delta,B} s_2{\downarrow_{\Delta,B}} \ldots}$$

By coercion, any term in canonical form that cannot be further rewritten via $\rightarrow_{R,B}$ is also considered to be a computation.

We define a *Maude step* from a given term s as any of the sequences $s \rightarrow^*_{\Delta,B} s{\downarrow_{\Delta,B}} \rightarrow_{R,B} t \rightarrow^*_{\Delta,B} t{\downarrow_{\Delta,B}}$ that head the non-deterministic Maude computations for s. Note that, for a canonical form s, a Maude step for s boils down to $s \rightarrow_{R,B} t \rightarrow^*_{\Delta,B} t{\downarrow_{\Delta,B}}$. We define $m\mathcal{S}(s)$ as the set of all possible *Maude steps* stemming from s in R. Finally, by *length*$(\mathcal{C})$ we define the number of Maude steps that are contained in the computation $\mathcal{C}$.

2.3 Generalization modulo Equational Theories

A key component of the semantic techniques proposed in this thesis is based on generalization (also known as anti-unification) [Plotkin, 1970]. A *generalization* of a pair of terms t_1, t_2 is a triple (g, θ_1, θ_2) such that $g\theta_1 = t_1$ and $g\theta_2 = t_2$. The triple (g, ϕ_1, ϕ_2) is the *least general generalization (lgg)*[2] of the pair of terms t_1, t_2, written $lgg(t_1, t_2)$, if it is the least general expression t such that both t_1 and t_2 are instances of t under appropriate substitutions. In other words, (i) (g, ϕ_1, ϕ_2) is a generalization of t_1, t_2 and (ii) for every other generalization (g', ψ_1, ψ_2) of t_1, t_2, g' is more general than g. For the free theory, the $\mathtt{lgg}$ of a pair of terms is unique up to variable renaming [Lassez et al., 1988].

The notion of least general generalization can be extended to work modulo order-sorted equational theories, where function symbols can obey any combination of associativity, commutativity, and identity axioms (including the empty set of such axioms) [Alpuente et al., 2009b, Alpuente et al., 2008, Alpuente et al., 2014d]. Unlike the untyped case, for a pair of terms t_1, t_2 there is generally no single lgg, due to order-sortedness or to the equational axioms. Instead, there is a finite, minimal, and complete set of lggs (denoted by $lgg_E(t_1, t_2)$) so that any other equational generalization has at least one of them as an instance. Given any element g of the set $lgg_E(t_1, t_2)$, we define the function π from $\mathcal{VPos}(g)$ to $\mathcal{Pos}(t_1)$ that provides an injective correspondence between (the position of) any variable in g and (the position of) the corresponding term in t_1; we need this because computing modulo equational axioms may cause the term structure of g to be different from both t_1 and t_2.

By $\widehat{lgg}_E(t_1, t_2)$, we denote the pair (G, π) where $G = (g, \phi_1, \phi_2)$ is arbitrarily chosen among those lggs in the set $lgg_E(t_1, t_2)$ that have fewer variables, and π is the corresponding position mapping from positions of g's variables to the relative subterms of t_1.

Example 1. *Let* $\mathtt{f}$ *be an associative and commutative symbol. Let* t_1 *and* t_2 *be terms such that* $t_1 = \mathtt{f(b, c, a)}$ *and let* $t_2 = \mathtt{f(d, a, b)}$*. Then, a possible lgg modulo the associativity and commutativity of* $\mathtt{f}$ *is* $(\mathtt{f(a, b, X)}, \{\mathtt{X/c}\}, \{\mathtt{X/d}\}) \in lgg_E(t_1, t_2)$*, where* $\mathtt{X}$ *is a variable. Note that both* t_1 *and* t_2 *are syntactically different from* $\mathtt{f(a, b, X)}$*, and the value* $\pi(3) = 2$ *indicates the subterm* $\mathtt{c}$ *of* t_1 *that is responsible for the mismatch with* t_2*.*

[2]Also known as *most specific generalizer (msg)* or *least common anti-instance (lcai)*.

Chapter 3

The Maude System and Language

Maude is a high-performance declarative language and system that naturally implements rewriting logic, which is a logic of change that supplements an extension of the membership equational logic by adding rewrite rules that are used to describe non-deterministic transitions between states.

This chapter summarizes the most relevant features of Maude. The chapter is organized as follows. Section 3.1 briefly summarizes the origins of Maude. Section 3.2 addresses some relevant syntactic, semantic, and operational technicalities of the current release of Maude, and Section 3.3 extends the discussion to Full Maude, which is the (augmented) implementation of Maude written in Maude itself.

3.1 Origins of Maude

According to [Clavel et al., 2015], Maude's birth dates back to the early '90s at the Logic and Specification Group inside SRI International in Menlo Park, California. After years researching on order-sorted equational logic [Goguen and Meseguer, 1992], José Meseguer proposed a new (sound and complete) logic called rewriting logic [Meseguer, 1992]. This new logic is an extension of the former order-sorted equational logic with the addition of new rules that model non-determinism, hence providing a general framework for unifying a wide variety of models of concurrency as well as pure declarative support for concurrent object-oriented programming [Clavel et al., 2015]. Following the path of order-sorted equational logic, which was implemented by defining a new programming language called OBJ3 [Goguen et al., 1988], rewriting logic led to the creation of its own language: Maude.

Although [Meseguer, 1992] already provided the syntax and operational and denotational semantics of Maude from the very begining, its first implementation would be delayed for a few years, just until the arrival of Steven Eker and Manuel Clavel to SRI International. In 1996, a preliminary version of Maude was presented at the International Workshop on Rewriting Logic and its Applications (WRLA) [Clavel et al., 1996], but only in 1999 the first version of Maude (i.e., Maude 1) was publicly released after being presented at the International Conference on Rewriting Techniques and Applications (RTA) [Clavel et al., 1999].

Full Maude [Durán, 1999, Clavel et al., 2007] is an extension of Maude written in Maude itself. The success of Full Maude as a test field for extending Maude itself led to the distinction between the two levels of Maude to avoid any possible ambiguity. Ever since there has been a distinction between Core Maude, which is the official Maude release implemented in C++, and Full Maude, which refers to the (also official) Maude implemented in Maude.

The last version of Maude is Maude 2.7.1, which includes as major novelties the efficient built-in implementation of variant-based unification and narrowing [Durán et al., 2016], and support for the external satisfiability modulo theories (SMT) solver CVC4 [Barrett et al., 2011].

3.2 Core Maude

This section provides a basic overview of Core Maude, which is the standard release of Maude implemented in C++, from its syntax to some interesting technicalities that are relevant to this thesis. For a more comprehensive overview, please refer to Part I of [Clavel et al., 2016].

3.2.1 Effective Parsing of Inputs

One of the many virtues of Maude is its flexiblity. Specifically, Maude allows users to define their own syntaxes, which can include mixfix operator declarations. This feature grants users the ability to specify a problem in countless different ways ranging from a more sober, traditional way to artistic notations that verge on the esoteric programming. Still, parsing in Core Maude is very effective thanks to a combination of different technologies that are used in stages. Roughly speaking, the internal mechanism that allows Maude to parse inputs comprises, in this order, the following procedures:

1. First of all, a flex/bison-based syntactic parser is executed, which interprets the input according to the basic syntax of Maude.

2. Then, a grammar generator creates a context-free grammar based on the user's signature and mixfix operators defined.

3. Finally, parser generator MSCP[1] [Quesada, 1997] produces a specific parser for that context-free grammar, which can be used to finish interpreting the input.

Though allowing mixfix declarations empowers the language capabilities, it also encourages the appearance of ambiguities. Nevertheless, Maude deals with them by using a mechanism based of precedence values and gathering patterns, which are explicitly assigned to an operator by means of the `prec` and `gather` attributes (see Section 3.2.8 of this document) or assigned with default values following the rules described in Sections 3.9.1 and 3.9.2 of [Clavel et al., 2016].

3.2.2 Basic Syntax

Modules, sorts, and operators in Maude are named by using identifiers, which are formed as a finite sequence of ASCII characters with the following restrictions:

- Characters '`{`', '`}`', '`(`', '`)`', '`[`', '`]`', '`,`', and the blank space are special in the sense that they break the sequence of characters into several identifiers.

- The backquote character '`` ` ``' is the escape character, which indicates that the next character does not break the sequence. Hence, it can only appear before any of the special characters mentioned above.

Moreover, when dealing with identifiers several limitations can arise, which are not discussed in this document (e.g., the use of the underscore character '`_`' when declaring operators). They are, however, in-depth explained in [Clavel et al., 2016].

[1]MSCP is named after the Maude Syntactic Constraint Propagation algorithm.

3.2.3 Modules

Modules in Maude are the top-level entities of specification and programming. They group all the necessary information to describe a system (by means of syntax declarations) and its properties (by means of statements). Core Maude distinguishes between two kinds of modules: functional modules and system modules. Roughly speaking, functional modules define data types and operations by means of equational theories while system modules specify rewrite theories.

Functional Modules

Functional modules define data types and operations on them by means of membership equational theories, which are assumed to be Church-Rosser and terminating, which ensures that the process of reducing a term by applying all possible equations to it eventually stops and its result is always the same for the same input term and program, regardless of the order of application of membership axioms and/or equations. Functional modules are declared by using the following scheme:

```
fmod ModuleId is
  ImportList SortSet SubsortDeclSet OpDeclSet
  MembAxSet EquationSet
endfm
```

where *ModuleId* is either the identifier of the (non-parameterized) module or the identifier of the (parameterized) module together with a list of parameter declarations, *ImportList* is a list of module importations[2], *SortSet* is a set of sorts declarations, *SubsortDeclSet* is a set of subsorts declarations, *OpDeclSet* is a set of operator declarations, *MembAxSet* is a set of (conditional) membership statements, and finally *EquationSet* is a set of (conditional) equational statements.

System Modules

System modules are rewrite theories where transitions between states are modeled by means of rules. Syntactically, system modules are but functional

[2]Since functional modules correspond to equational theories, they do not contain rules. Hence, functional modules can only import other functional modules or functional theories, but not system modules or system theories.

modules with the addition of rule statement declarations. Therefore, declaration of system modules is performed by using the following analogous scheme:

```
mod ModuleId is
   ImportList SortSet SubsortDeclSet OpDeclSet
   MembAxSet EquationSet RuleSet
endm
```

where *ModuleId*, *ImportList*, *SortSet*, *SubsortDeclSet*, *OpDeclSet*, *MembAxSet*, and *EquationSet* have the same meaning as in functional modules declarations, and *RuleSet* is a set of (conditional) rule statements.

Module Importation

Maude allows users to define module hierarchies[3], which facilitate the reusability of components and the understanding and debugging of programs by keeping modules in a relatively small size. Module hierarchies are accomplished by importing modules into other modules as submodules, being the submodule relation transitive.

In Maude, modules can import other modules in three different ways by using the `protecting`, `extending`, and `including` keywords (or their respectively abbreviations `pr`, `ex`, and `inc`) followed by a module expression:

```
pr ModuleExpression .
ex ModuleExpression .
inc ModuleExpression .
```

where *ModuleExpression* can be the identifier of a module or the result of a module operation such as renaming of a module, instantiation of a parameterized module, summation of modules, etc.

The main differences between all three importation modalities can be summarized as follows.

- The `protecting` mode indicates that if a module A imports a module B, for any sort s of B, A should not add any new ground terms of sort s in canonical form. Moreover, A should not have any declaration that makes two distinguishable terms of B indistinguishable in A.

[3]A module hierarchy is an acyclic graph of module importations.

- The `extending` mode has weaker constraints than the previous mode, since only the second constraint (i.e., the distinction of two previously indistinguishable terms) is assumed to hold.

- Finally, the `including` mode poses no restrictions regarding these aspects.

Still, it is important to note that Maude does not check the satisfaction of module importation constraints per se and it is up to the user to ensure that they are indeed satisfied.

Predefined Modules

Maude has a standard library of predefined modules (see Figure 3.1) that define commonly used data types (e.g., boolean, string, natural, integer, etc.) and typical operations over them. Furthermore, some common parameterized collections and associations of data types such as lists, sets, arrays, and mappings are also defined.

Maude's predefined modules are declared at the *prelude*, which is automatically loaded into the system at the beginning of each session so that users can import them into their modules. By default, the predefined module `BOOL` is included as a submodule in any user-defined module, while the inclusion of the rest of predefined modules has to be explicitly stated.

Parameterized Modules

In Maude, parameterized modules are (system or functional) modules with one or more parameters, each of which being expressed by means of a theory. They are declared by using the same module syntactic schemes previously described. For example, a parameterized system module can be declared as follows:

```
mod ModuleId{ ParameterDeclList } is ... endm
```

where *ModuleId* is the identifier of the module and *ParameterDeclList* is a list of parameter declarations of the form $X_1 :: T_1$, ..., $X_n :: T_n$, with $n \geq 1$, where each tuple $X_i :: T_n$ is a parameter declaration consisting of an identifier X_i and an expression T_i that returns a theory.

As for their instantiations, parameterized modules are accomplished by using theories and views, which are succinctly addressed in Sections 3.2.4

and 3.2.5 of this manuscript. A much more extensive description about the parameterizing capabilities of Maude is available in Section 6.3 of [Clavel et al., 2016].

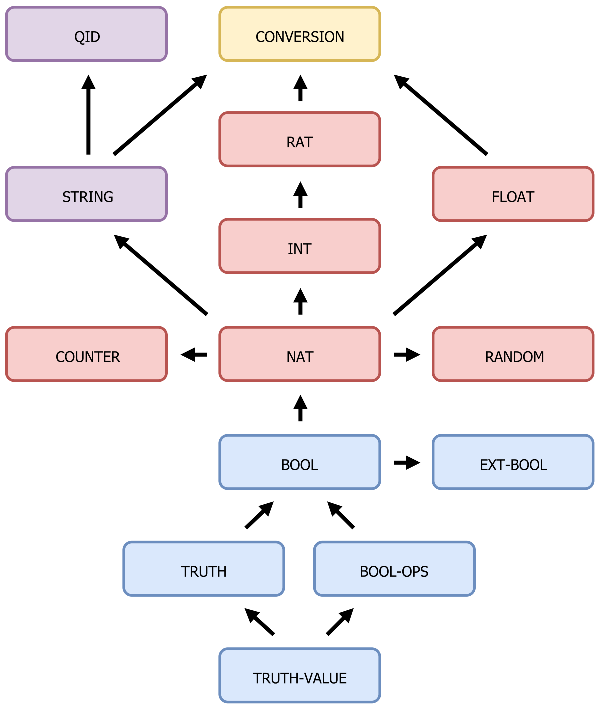


Figure 3.1: Hierarchy of predefined modules in Maude.

3.2.4 Theories

Theories in Maude are the means to specify the syntactic and semantic properties that actual parameters must satisfy for the instantiation of parameterized modules. Just like modules, there are two types of theories: functional theories and system theories.

Functional Theories

Functional modules and functional theories are both membership equational logic theories. The most significant difference though is that functional theories do not need to be Church-Rosser and terminating. However, since theories can be executed in the same way as modules, the set of executable equations and membership axioms of a theory must preserve these two properties. Consequently, non-executable[4] equations do not need to satisfy the two properties, but they must be identified by using the `nonexec` attribute in their respectively declarations. Moreover, non-executable equations and membership axioms may have right-hand sides and conditions that include variables not appearing in their corresponding left-hand sides. Non-executable statements can also have left-hand sides consisting of a single variable, which is not usually permitted in standard term rewrite theory.

Similarly to functional modules, functional theories are declared by using the following syntax:

```
fth TheoryId is
  ImportList SortSet SubsortDeclSet OpDeclSet
  MembAxSet EquationSet
endfth
```

where *TheoryId* is the identifier of the theory and *ImportList*, *SortSet*, *SubsortDeclSet*, *OpDeclSet*, *MembAxSet*, and *EquationSet* have the same meaning as in their module declaration counterparts. Note that functional theories can not be parameterized, hence the need for a simple identifier as a header.

System Theories

Analogously to system modules, system theories specify rewrite theories. For execution purposes, system theories are treated just as system modules with the exception of the non-executable rules, equations, or membership axioms labeled with the `nonexec` attribute, which are ignored by the rewriting and narrowing mechanisms of Maude. Nevertheless, execution of non-executable statements can be *manually* achieved at the metalevel by following user-defined strategies. As for their syntax, system theories are specified by using the following scheme:

```
th TheoryId is
  ImportList SortSet SubsortDeclSet OpDeclSet
  MembAxSet EquationSet RuleSet
endth
```

[4]Non-executable equations can be executed in a controlled manner at the metalevel, but they are ignored by the Maude rewrite engine.

where *TheoryId* is the identifier of the theory and *ImportList*, *SortSet*, *SubsortDeclSet*, *OpDeclSet*, *MembAxSet*, *EquationSet*, and *RuleSet* have the same meaning as in the declaration of system modules. Similarly to functional theories, note that system theories can not be parameterized either.

Theory Importation

Theories use the same importation mechanism as modules, with the exception that a theory can only import another theory in `including` mode.

3.2.5 Views

Views are used to describe the interpretation of a source theory in the target theory or module, that is, to specify the mapping (of sorts and operators) from the source theory to the target theory. Since there can be multiple interpretations that ensure the satisfaction of requirements of a source theory by a target one, there can be multiple views for the same pair of (source and target) theories, each one describing a particular interpretation. Views can be defined according to the following syntax:

```
view ViewId from SourceModuleExpression to TargetModuleExpression is
  SortMappingSet OpMappingSet
endv
```

where *ViewId* is the identifier of the view, *SourceModuleExpression* is an expression that yields the source theory, *TargetModuleExpression* is an expression that yields the target theory or module, and *SortMappingSet* (resp. *OpMappingSet*) is the set of mappings that relate sorts (resp. operators) in the source theory to sorts (resp. operators) in the target module or theory. Likewise, mappings of sorts from one theory to another theory or module are specified as follows:

```
  sort SourceId to TargetId .
```

where *SourceId* is the identifier of the source theory and *TargetId* is the identifier of the target theory or module. As for operators, their mappings can be specified in three different ways:

```
  op SourceId to TargetId .
  op SourceId : TypeList -> Type to TargetId .
  op SourceTerm to term TargetTerm .
```

where *SourceId* (resp. *TargetId*) corresponds to the identifier of the source (resp. target) operator, *TypeList* is the (sorted) list of types of the arguments of a given source operator, *Type* is the coarity (i.e., the type) of the same source operator, and *SourceTerm* (resp. *TargetTerm*) is the source (resp. target) operator. Note that in the first case all operators with the same name are affected, while in the second case operators matching a given arity and coarity and their entire family of overloaded operators are affected.

3.2.6 Sorts, Subsorts and Kinds

Sorts are types, in particular, the types of the data and operations defined in a specification. Though Maude has some predefined sorts declared (e.g., `Nat`, `Float`, `Char`, `String`, `Bool`, etc.) in general, types can be declared with relative freedom. Declaration of sorts can be done either separately or jointly by using the following syntactic constructs:

`sort` $SortId_1$ `.`
`sorts` $SortId_1$ ... $SortId_n$ `.`

where $n \geq 1$ and each sort of the form $SortId_i$ is the identifier of a newly defined type. Sorts can also be partially ordered[5] by means of subsort relations, which are defined as follows:

`subsort` $SortId_1$ `<` $SortId_2$ `.`
`subsorts` $SortId_1$ ... $SortId_i$ `<` ... `<` $SortId_j$ ... $SortId_k$ `.`

where $1 \leq i < j \leq k$. In the first case, where a single subsort relation is defined, $SortId_1$ is declared a subsort of $SortId_2$, whereas in the second case, where multiple subsort relations are defined, sorts $SortId_1$ ... $SortId_i$ are declared subsorts of sorts $SortId_j$ ... $SortId_k$.

A side-effect of partially ordering the set of sorts of a specification is that they are arranged into disjoint, ordered sets called connected components, which define equivalence classes named kinds. In Maude, kinds are named by encasing the identifier of their top sort (if it is unique) or the list of all their top sorts (if there are more than one) with brackets. For example, Figure 3.2 shows the (six) connected components, namely `[DecFloat]`,

[5]To define a partial order among the set of sorts, the declaration of subsorts must not contain cycles.

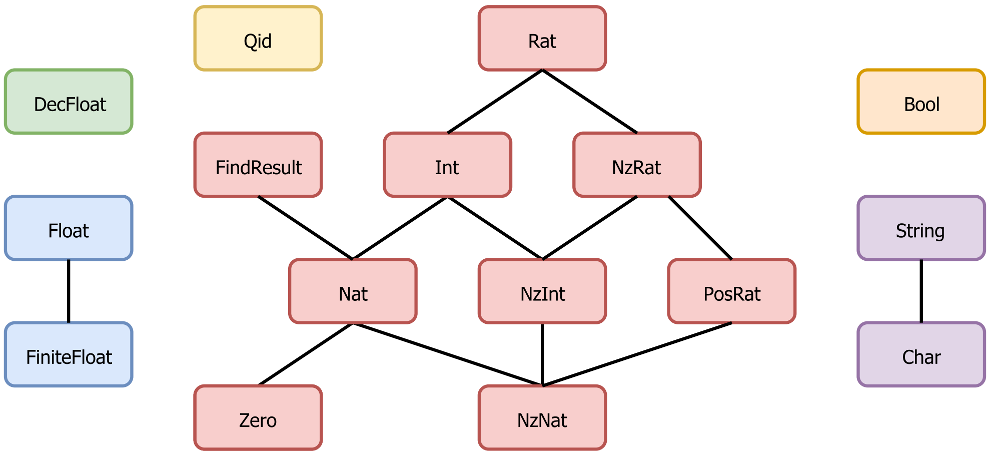


Figure 3.2: Partial order of sorts declared in the prelude of Maude.

`[Float]`, `[Qid]`, `[Rat,FindResult]`, `[Bool]`, and `[String]`, resulting from the sort and subsort declarations inside the predefined modules of Figure 3.1. Note that all those connected components have unique top sorts except for `[Rat,FindResult]`, which has two, namely `Rat` and `FindResult`.

3.2.7 Variables

Variables in Maude are specified inline by concatenating an identifier, a colon, and the sort or kind in which they are constrained to range over (e.g., `X:Nat` declares a variable named `X` of sort `Nat` and `Y:[Nat]` declares a variable named `Y` of kind `[Nat]`).

Variables can also be declared inside modules[6] so that they can be used in the declaration of memberships, equations, or rules. Declaration of variables inside modules is thus accomplished by using the following constructs:

var $\mathit{VariableId}_1$: *Sort* .
var $\mathit{VariableId}_1$: *Kind* .

vars $\mathit{VariableId}_1$ ... $\mathit{VariableId}_n$: *Sort* .

[6]Note that the declaration of variables inside modules is just *syntactic sugar*, since they are not part of modules (see Section 3.2.3). Hence, they are translated into inline declarations at the metalevel.

vars *VariableId*$_1$ ... *VariableId*$_n$: *Kind* .

where $n > 1$, each *VariableId*$_i$ is the name of a new variable, and *Sort* and *Kind* are respectively the sort and kind associated with the variable.

Finally, note that variables declared inside modules can be reused in the declaration of different equations, membership axioms, or rules of the same module, but then they are considered to be different variables even though they share the same identifier and sort or kind. Nevertheless, multiple occurrences of a given identifier of a variable inside a (conditional) statement refer to the same variable.

3.2.8 Operators

Similarly to sorts or variables, operators can be declared either individually or in groups, provided they share arity and coarity.

Operators with zero arity are named constants[7], and their syntax is as follows:

op *OperatorId*$_1$: -> *Sort* [*OperatorAttrs*] .
ops *OperatorId*$_1$ ... *OperatorId*$_n$: -> *Sort* [*OperatorAttrs*] .

where $n \geq 1$, each *OperatorId*$_i$ is the name of a new constant, *Sort* is the respectively coarity, and *OperatorAttrs* is an optional list of (operator) attributes. Furthermore, operators with one or more arguments can be declared by using the following scheme:

op *OperatorId*$_1$: *Sort*$_1$ ... *Sort*$_n$ -> *Sort* [*OperatorAttrs*] .
ops *OperatorId*$_1$ ... *OperatorId*$_m$: *Sort*$_1$ ... *Sort*$_n$ -> *Sort* [*OperatorAttrs*] .

where $m \geq 1$, $n \geq 1$, each *OperatorId*$_i$ is the name of an operator, each *Sort*$_i$ is the type of the *i-th* argument of the operator, *Sort* is the coarity, and *OperatorAttrs* is also an optional list of (operator) attributes.

Operator Attributes

Maude allows the user to provide additional syntactic, semantic, or operational information about an operator by introducing some predefined attributes in their dec-

[7]Note that constants in Maude do not share the traditional concept of immutability given in most imperative languages, since they can be rewritten into other terms by means of the application of appropriate equations or rules.

larations. The list of admissible operator attributes that are relevant to this thesis together with a brief description is as follows:

- `assoc` indicates that the operator has the associativity property.
- `comm` indicates that the operator has the commutativity property.
- `idem` indicates that the operator has the idempotency property.
- `id:` *Term* indicates that the operator has the identity property, with *Term* being the identity element.
- `left id:` *Term*, indicates that the operator has the left identity property, with *Term* being the left identity element.
- `right id:` *Term*, indicates that the operator has the right identity property, with *Term* being the right identity element.
- `iter`, short for iterated operator, allows very large stacks of unary operators to be easily introduced or manipulated. For example, given an iterated, unary operator `f`, the term `f(f(f(X:Nat)))` is equivalent to `f^3(X:Nat)`.
- `ctor`, indicates that the operator is a constructor[8].
- `poly` (*SortSeq*) stands for *polymorphic* operator, where *SortSeq* is a (non-empty) sorted[9] sequence of natural numbers, each of them identifying the position of a *universal* argument in the operator declaration (i.e., an argument that is not constrained to a particular sort and that is declared to be of the special sort `Universal`). For example, `op if_then_else_fi : Bool Universal Universal -> Universal [ poly (2 3 0) ]` declares a polymorphic 3-ary operator, namely `if_then_else_fi` whose second and third arguments are of sort `Universal`, as well as its coarity.
- `format` (*InsWord*), where *InsWord* is an instruction word based on an alphabet consisting of symbols '`d`' (for default spacing), '`+`' (for increasing the indent counter), '`-`' (for decreasing the indent counter), '`s`' (for space), '`t`' (for tab),'`i`' (for the number of spaces determined by the indent counter), and '`n`' (for newline), allows to define how to format the color and white space of terms when printing them (see Section 4.4.5 of [Clavel et al., 2016]).

[8]Note that Maude does not check for the validity of this claim. Nevertheless, it can be easily checked by using the Maude's Sufficient Completeness Checker (SCC) [Hendrix et al., 2005].

[9]The order established in the operator declaration is as follows: the first argument is associated with position one (and so on) and the coarity, which is always stated in the last position, is associated with position zero.

- `prec` *N*, where *N* is a natural number that states the precedence value of the operator. This attribute allows (in combination with the `gather` attribute) to reduce the number of possible ambiguities when parsing.
- `gather` (*SortSeq*), where *SortSeq* is a sequence of symbols '`E`', '`e`', or '`&`', each of which restricts the precedence value of a term to be admissible as a corresponding argument. Specifically, '`E`' indicates that the argument must have a precedence value lower than or equal to the operator, '`e`' that the precedence value of the argument must be strictly lower, and '`&`' allows any precedence value for a term to be admissible as the argument.
- `memo`, which provides an improvement in efficiency by instructing Maude to *memoize* the results of equational simplification for those subterms that have the operator at their respectively top positions, so that the reduction result can be swiftly and directly retrieved in subsequent computations.
- `metadata` *Str*, where *Str* is a term of sort `String`, allows users to associate useful information with the operator.
- `special` is reserved to bind built-in operators that are defined in the prelude by using Maude syntax with their appropriate C++ code and functionality.

From a syntactic point of view, (operator) attributes can be combined with absolute freedom. However, from a semantic point of view, there are some restrictions and side-effects that must be considered. Following are some of relevance:

- Arguments of binary operators that are declared with either `assoc`, `comm`, `id`, or `idem` must belong to the same kind.
- The attribute `idem` can not be used in combination with `assoc`. However, Maude enforces compliance with this constraint by ignoring the former attribute (while keeping the latter) when needed.
- Only the first identity attribute (i.e., `id:`, `left id:`, or `right id:`) appearing in an operator declaration is considered. Subsequent identity occurrences, although different, are simply ignored.
- Combining either `left id:` or `right id:` with `comm` converts the former two attributes into `id:`.
- Membership axioms can interact in undesirable ways with operators decorated with the `assoc` or `iter` attributes (see Section 14.2.8 and 14.2.9 of [Clavel et al., 2016]).
- It is recommended not to modify operators declared to be `special` in any way, since it might produce serious stability problems.

3.2.9 Statements

Statements in Maude are either membership axioms, equations, or rules, all of which may be conditional or not. In the following there are described all three types of statements together with a list of admissible (statement) attributes and conditions.

Membership Axioms

Membership axioms specify that terms that are matched by a given pattern must have a given sort. They can be unconditional as well as conditional. Unconditional membership axioms are declared by using the following constructs:

```
mb Term : Sort [ StatementAttrs ] .
mb [ Label ] : Term : Sort [ StatementAttrs ] .
```

where *Term* is a pattern, *Sort* is the sort associated to terms that are matched by *Term*, *StatementAttrs* is an optional set of statement attributes (shown below), and *Label* is the identifier of the membership statement. As for conditional membership axioms, they are declared in an analogous manner:

```
cmb Term : Sort if EqCond [ StatementAttrs ] .
cmb [ Label ] : Term : Sort if EqCond [ StatementAttrs ] .
```

where *Term*, *Sort*, and *Label* have the same meaning as in their unconditional counterparts and *EqCond* is an equational condition.

Finally, remember that membership axioms can interact in undesirable ways with operators having the `assoc` or `iter` attributes (see Section 14.2.8 and 14.2.9 of [Clavel et al., 2016]). Moreover, the use of a single variable pattern in a membership statement may lead to non-termination.

Equations

The most relevant characteristic of equations in Maude is that they are required to be Church-Rosser and terminating[10]. Unconditional equations are declared by using the following schemata:

```
eq Term_lhs = Term_rhs [ StatementAttrs ] .
```

[10]Except for those equations holding the `nonexec` attribute, which are simply ignored by the internal simplification mechanism of Maude.

```
eq [ Label ] : Term_lhs = Term_rhs [ StatementAttrs ] .
```

where *$Term_{lhs}$* and *$Term_{rhs}$* are respectively the left-hand side and right-hand side terms[11] of the equation, *StatementAttrs* is an optional list of (statement) attributes, and *Label* is the identifier of the equational statement.

Equations can also be conditional, which are declared as follows:

```
ceq Term_lhs = Term_rhs  if EqCond [ StatementAttrs ] .
ceq [ Label ] : Term_lhs = Term_rhs  if EqCond [ StatementAttrs ] .
```

where *$Term_{lhs}$*, *$Term_{rhs}$*, *StatementAttrs*, and *Label* have the same meaning as in the declaration of unconditional equations, and *EqCond* is an equational condition.

Rules

Rules are used to model (non-deterministic) transitions between states in a system. Hence, unlike equations, rules are not required to be confluent and terminating. The syntax for declaring unconditional rules is as follows:

```
rl Term_lhs => Term_rhs [ StatementAttrs ] .
rl [ Label ] : Term_lhs => Term_rhs [ StatementAttrs ] .
```

where *$Term_{lhs}$* and *$Term_{lhs}$* are respectively the left-hand side and right-hand side terms of the rule, *StatementAttrs* is an optional list of statement attributes, and *Label* is the identifier of the rule statement.

Similarly, conditional rules are declared by using the following syntax:

```
crl Term_lhs => Term_rhs if Cond [ StatementAttrs ] .
crl [ Label ] : Term_lhs => Term_rhs if Cond [ StatementAttrs ] .
```

where *$Term_{lhs}$*, *$Term_{rhs}$*, *StatementAttrs*, and *Label* have the same meaning as in the declaration of unconditional rules and *Cond* is a condition.

Statement Attributes

Similarly to operators, statements can be further defined by adding attributes to their declarations. As for Maude 2.7.1, there are six admissible statement attributes:

[11]Note that, since equations define equalities, both *$Term_{lhs}$* and *$Term_{rhs}$* are required to belong to the same equivalence class (i.e., the same kind).

- `label` *Qid*, which facilitates debugging and tracing by associating a statement with an identifier. Note that statements can also be labeled by declaring them using the *sugared* syntactic constructs discussed above. Nevertheless, in case of discrepancy between the identifier stated by the *sugared* declaration of the statement and the identifier stated by this attribute, the former prevails.

- `metadata` *Str*, where *Str* is a term of sort `String`, allows users to associate useful information with the statement.

- `nonexec`, which states that the rewriting and narrowing mechanisms of Maude must ignore the statement. Nevertheless, as mentioned before, non-executable statements can be executed at the metalevel in a controlled manner.

- `owise`, short for *otherwise*, alters the (non-established) order of application of equations while reducing a term to its canonical form so that those equations with the `owise` attribute are always considered as the last option. Therefore, this attribute can only be used in the declaration of equations. Moreover, note that the use of `owise` directly affects the Church-Rosser and termination properties of equational definitions, which must be preserved in any case.

- `print` Str_1 Var_1, ..., Str_n Var_n where $n \geq 1$, each Str_i is a term of sort `String`, and for all i in $1..n$ Var_i is a variable in the domain of the statement (e.g., `rl f(X,Y) => g(X,Y) [ print "VarX = " X, "VarY = " Y ]`), prints useful information about the values of variables `X` and `Y` whenever the statement is executed.

- `variant`, which is used to identify those equations that are to be used for variant generation or variant-based unification. Note that, because of the restrictions imposed by the current implementation of variant generation in Maude, this attribute can only be used in the declaration of non-conditional equations. Moreover, it is also incompatible with the use of the `owise` attribute.

Statement Conditions

Let $\mathcal{R} = (\Sigma, E, R)$ be a conditional rewrite theory, with $E = \Delta \cup B$, where Δ is a set of conditional equations and membership axioms and B is a set of equational axioms (e.g., associativity, commutativity, and identity) associated with some binary operators in the signature Σ. Let σ be the matching substitution computed in the application of a conditional statement to a term. Then, an admissible condition for such statement is either a single statement or a conjunction of statements specified by using the (associative) connective $\wedge$ as follows:

$$Cond_1 \wedge \ldots \wedge Cond_n$$

where each statement $Cond_i$ is either:

- an ordinary equation of the form $Term_m = Term_p$, which only evaluates to `true` if $(Term_m\sigma){\downarrow_{\Delta,B}} = (Term_p\sigma){\downarrow_{\Delta,B}}$,
- an abbreviated equation $Term$, with $Term$ a term in the `[Bool]` kind, which abbreviates the equation $Term =$ `true`,
- a matching equation of the form $Term_p := Term_m$, which evaluates to `true` if $Term_p\sigma$ matches $(Term_m\sigma){\downarrow_{\Delta,B}}$. Note that, by matching both terms, the fresh variables of $Term_p$ (i.e., those variables that do not appear in the left-hand side of the conditional statement) become instantiated. Also, for this match to decide the equality with $(Term_m\sigma){\downarrow_{\Delta,B}}$, then $Term_p$ must be a Δ-pattern, that is, a term t such that for every substitution ϕ, if $x\phi$ is a canonical form with respect to Δ modulo B for every $x \in Dom(\phi)$, then $t\phi$ is also a canonical form with respect to Δ modulo B.
- or a rewrite expression $Term_m \Rightarrow Term_p$ that evaluates to `true` if there exists a computation $Term_m\sigma \rightarrow^*_{R/E} Term'_m$ such that $Term_p\sigma$ matches $Term'_m$. Note that, as in matching equation conditions, the possible fresh variables of $Term_p$ become instantiated if the rewrite condition finally evaluates to `true`.

It is important to remark that the first three types of conditions are equational conditions, which can be freely used in the declaration of all types of conditional statements, whereas the last type of condition is a rewrite expression, which can only be used in the declaration of conditional rules. Moreover, although conditions can be listed in any order regardless of their type, operationally speaking the order is very much relevant, since the evaluation of such conditions is performed from left to right, and so is done the binding of possible fresh variables appearing in matching or rewrite conditions, which may affect the final result.

3.3 Full Maude

For the sake of maximizing its expressiveness, Maude was endowed with metaprogramming capabilities from the very beginning. In practice, Full Maude is much more than just the implementation of Core Maude in the Maude language. It is an extended version that has been proved to be notably useful as a test field in the development of Maude by allowing the developers to define new features with reasonably low cost, which are then ported to Core Maude when they become mature enough.

The rest of this section briefly addresses the main features of Full Maude in regard to this thesis, from the basic insights to the object-oriented improvements it offers.

3.3.1 Basic Insights

Here are presented two of the most relevant characteristics of Full Maude, i.e., the use of the loop module of Core Maude to mimic the interactive capabilities of the Maude interpreter and the module database, which also mimics the persistence of the original system.

Using the Loop

To mimic the interaction capabilities of the Maude system, Full Maude makes use of the `LOOP-MODE` module of Core Maude, which provides support for basic input/output interaction by means of a generic read-eval-print loop. As simple as powerful, the `LOOP-MODE` module is actually part of Maude's prelude and is defined as follows:

```
mod LOOP-MODE is
  protecting QID-LIST .
  sorts State System .
  op [_,_,_] : QidList State QidList -> System
   [ ctor special (
    id-hook LoopSymbol
    op-hook qidSymbol (<Qids> : ~> Qid)
    op-hook nilQidListSymbol (nil : ~> QidList)
    op-hook qidListSymbol (__ : QidList QidList ~> QidList))
   ] .
endm
```

As shown above, `LOOP-MODE` consists of the declaration of a single, special ternary operator `[_,_,_]` that holds all the data needed for such interaction. This operator is called the *loop object* and its arguments are, in this order, the input stream, the state of the loop, and the output stream. There are some limitations though. Currently, the input stream is inevitably linked to Maude's terminal, thus leading to some ambiguity problems. Specifically, the Maude system has to distinguish which inputs should be interpreted by Core Maude and which by the Maude program that has activated the loop (e.g., Full Maude). To workaround this problem, Maude requires an active distinction between inputs by forcing to enclose with parenthesis those inputs that are to be interpreted within the loop. However, this clever solution implies losing the advantages that the highly efficient parser of Maude offers and painfully slows the parsing process for very large inputs such as execution traces. Nevertheless, this issue can be easily solved by implementing a small input framework that is discussed in Section 7.3.2 of this thesis.

Module Database

By taking advantage of the above mentioned *loop object*, Full Maude is capable to maintaining a *semi-persistent*[12] database in which modules can be easily stored for later access. Equally interesting is that modules loaded in Core Maude prior to the execution of Full Maude can also be freely accessed from within the later, which allows users to combine both Core and Full Maude's declared modules to specify their programs.

3.3.2 Object-Oriented Programming

Object-oriented programming is a well known programming paradigm that aims to facilitate the modeling of systems by providing some conceptual advantages similar to those of human reasoning. In the following, the section presents the most relevant characteristics that Full Maude provides in order to easily implement object-oriented, concurrent systems.

Object-Oriented Modules

In addition to supporting both functional and system modules, Full Maude extends Core Maude with a new type of modules, namely object-oriented modules. This extension is just syntactic sugar, since object-oriented modules are internally transformed into system modules for execution purposes. Still, object-oriented modules provide interesting conceptual advantages to users when defining object-oriented systems. The declaration of object-oriented modules is performed by means of the following syntactic scheme:

```
omod ModuleId is
  ImportList SortSet SubsortDeclSet
  ClassDeclSet SubclassDeclSet OpDeclSet MsgDeclSet
  MembAxSet EquationSet RuleSet
endm
```

[12]The information enclosed in the database is inevitably lost once the *loop* is broken and the corresponding *loop object* destroyed.

where *ModuleId* is either the identifier of the module or (in case of parameterized modules) an expression consisting of an identifier together with a list of parameter declarations, *ImportList* is a list of module importations, *SortSet* is a set of sorts declarations, *SubsortDeclSet* is a set of subsorts declarations, *ClassDeclSet*, is a set of class declarations, *SubclassDeclSet*, is a set of subclass declarations, *OpDeclSet* is a set of operator declarations, *MsgDeclSet*, is a set of message declarations, *MembAxSet* is a set of (conditional) membership statements, *EquationSet* is a set of (conditional) equational statements, and finally *RuleSet* is a set of (conditional) rule statements.

Objects

An object in Full Maude is a term of the form `<` *Oid* `:` *Cid* `|` $Attr_1$, ..., $Attr_n$ `>`, with $n \geq 0$, where *Oid* is the identifier of the object, *Cid* is the class of the object, and each $Attr_i$ is an attribute, a tuple of the form *AttrName* `:` *AttrValue*.

As mentioned above, object-oriented modules offer a more convenient syntax and conceptual advantages, but they are internally transformed into system modules for execution purposes. A side effect of this translation is that positions in object terms can be easily misinterpreted since each attribute *AttrName* of sort *AttrSort* is automatically translated into a term of sort `Attribute` by dynamically introducing *ad-hoc* unary operators (one per attribute) of the form

```
AttrName`:_ : AttrSort -> Attribute
```

as illustrated in the following example.

Example 2. *Consider the following Full Maude object*

```
< 'A1 : EconomyCar | available : true , rate : 30 >
```

that models a car with identifier `'A1`*, class* `EconomyCar`*, and two attributes: a Boolean attribute* `available` *that indicates whether the car is available for renting, and an integer attribute* `rate` *that stores the rental price per day.*

Intuitively, we can wrongly state that the value `true` *of attribute* `available` *appears at position* 3.1.2 *of the term but its position is* 3.1.1.

As pointed out, object-oriented modules in Full Maude are translated into Core Maude system modules for execution purposes. For this example in particular, the attributes `available` *and* `rate` *are automatically translated by using the following 1-arity operator declarations that are dynamically created by Full Maude:*

```
op 'available`:_ : 'Bool -> 'Attribute [ gather('&) ] .
op 'rate`:_ : 'Nat -> 'Attribute [ gather('&) ] .
```

A graphical, meta-level[13] *representation of the transformed term is depicted in Figure 3.3. Note that the value* `true` *of the attribute* `available` *is addressed by position* 3.1.1*, which is not obvious by only looking to the source-level representation of the original term.*

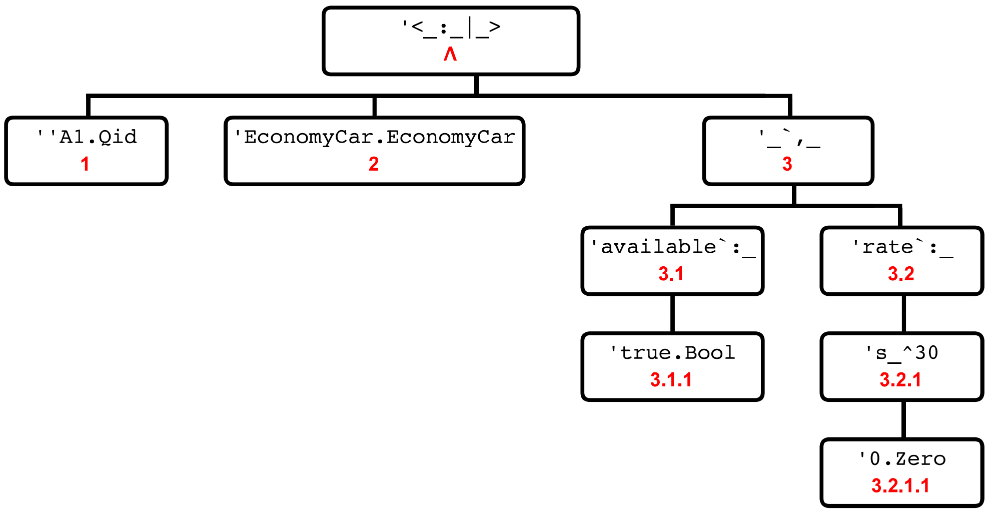


Figure 3.3: Positions of the car object of Example 2.

Classes and subclasses

Classes are defined by using the keyword `class`, followed by the name of the `class`, a bar, and a list of attribute declarations separated by commas:

`class` *ClassName* | *AttrName*$_1$: *Sort*$_1$, ... , *AttrName*$_n$: *Sort*$_n$.

Class names are considered to be a particular case of sort names. Therefore, class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration is an expression of the form

`subclass` *ClassName*$_1$ < *ClassName*$_2$.

[13]By using the meta-level notation, we can easily and unequivocally identify the arity of each operator, with the underscores indicating the exact place of its arguments in the mixfix notation.

where *ClassName*$_1$ and *ClassName*$_2$ are the names of the classes. Moreover, multiple inheritance is also supported, allowing a class to be defined as a subclass of several classes.

Messages

Messages are the elements that provide communication between objects. Messages do not have a fixed syntax, which is usually defined by the user. Their only requirement is that the first argument of a message is the object identifier of its destination object. Messages are thus declared by using the `msg` keyword as follows.

`msg` *MessageId* : *ObjectId*$_{dest}$ *Sort*$_0$ ... *Sort*$_n$ `-> Msg .`

where $n \geq 0$, *MessageId* is the name of the message, *ObjectId*$_{dest}$ is the object identifier of the destination object, and *Sort*$_0$ ... *Sort*$_n$ is a (possibly empty) list of sorts.

Chapter 4

Inspection of Rewriting Logic Computations

Dynamic analysis is crucial for understanding the behavior of large, often complex, software systems. Dynamic information is typically represented using execution traces whose analysis is almost impracticable without adequate tool support. Existing tools for analyzing large execution traces commonly rely on a set of visualization techniques that facilitate the exploration of the trace content. Common capabilities of these tools include stepping the program execution while searching for particular components. Program animation or *stepping* refers to the very common debugging technique of executing code one step at a time, allowing the user to inspect the program state and related data before and after the execution step. This allows the user to evaluate the effects of a given statement or instruction in isolation and thereby gain insight into the program behavior.

This chapter introduces an interactive program animation technique for the understanding and debugging of rewriting logic computations together with an instrumentation technique that aims to uncover equational and axiomatic steps that are usually hidden within Maude's rewrite machinery. The chapter is organized as follows. Section 4.1 introduces the two running examples that are used in the thesis to facilitate the understanding of the main notions and demonstrations. Section 4.2 presents a technique for instrumenting Maude steps that uncovers equational simplification and algebraic axiomatic steps and greatly facilitates the understanding of Maude executions. Section 4.3 formalizes an exploration technique that interactively constructs computation trees stemming from an initial input term. Finally, Section 4.4 shows an *animated* debugging session that exploits the techniques previously discussed in order to detect and locate possible program misbehaviors.

4.1 The Running Examples

Since the techniques that have been developed in this thesis can be applied to both Core Maude and object-oriented, Full Maude programs, two running examples, one of each aforementioned category, are introduced in this section and used throughout the thesis.

4.1.1 A Stock Exchange System

The first example consists of a (faulty) rewrite theory written in Core Maude that specifies a stock exchange concurrent system in which traders operate on stocks via limit orders, that is, orders that set the upper bound (price *limit*) at which traders want to buy stocks. The system is modeled primarily by means of the `STOCK-EXCHANGE` system module, which in turn imports parameterized and functional auxiliary modules and establishes a rather intricate hierarchy of modules and views. For the sake of clarity, Figure 4.1 focuses on the most relevant parts of the system. Nevertheless, the complete Maude specification of the stock exchange model can be found in Appendix A.

In `STOCK-EXCHANGE`, when the stock price equals or drops below the price limit `L`, the associated order is *opened* and the trader buys the stocks at the current stock price. An order is automatically *closed* and the associated stocks are sold either (a) when the current stock price `P` exceeds the purchase price limit `L` plus a predetermined *profit target* `PT` (i.e., $P - L \geq PT$), or (b) when $L - P$ exceeds a predetermined *stop loss* `SL` (i.e., $L - P \geq SL$).

Within the system model, variable names are fully capitalized, while names that begin with the symbol `'` are constant identifiers for traders, stocks and orders. System states have the form `R : SS | TS | OS`, where `R` is a natural number (called round) that models the market time evolution, and `SS`, `TS`, and `OS` are sets[1] of stocks, traders, and orders, respectively.

Stocks are modeled as terms `st(SID,P)` with `SID` being the stock identifier and `P` being the current stock price. Traders are modeled as `tr(TID,C)`, where `TID` is the trader identifier and `C` is the trader's available capital. We consider two classes of traders: premium traders and ordinary (or non-premium) traders. Premium traders are allowed to buy even if they run out of capital. Premium traders are identified by the conditional membership axiom `premT` (see Figure 4.1) that simply checks whether the trader identifier belongs to the (hard-coded) list `PreferredTraders`, which in this example just contains the premium trader `'T2`.

Orders are specified by terms of the form `ord(OID,TID,SID,L,PT,SL,ST)`, which record the order identifier `OID`, the trader identifier `TID`, the stock identifier `SID`, the stock price limit `L`, the profit target `PT`, the stop loss `SL`, and the order status `ST` (which can be either `open` or `closed`). For simplicity, an order allows only a single stock to

[1]To specify sets of `X`-typed elements, the Maude parameterized sort `Set{X}` is instantiated, which defines sets as associative, commutative, and idempotent lists of elements that are simply written as $(e_1, \ldots, e_n)$. The empty set is denoted by the constant symbol `empty`.

```
cmb [premT] : tr(TID,C) : PremiumTrader if TID in PreferredTraders .

eq [updP] : updP(R,S,(st(SID,P),SS)) =
    if (rndDelta(R * S) rem 2) == 0
    then st(SID,S + rndDelta(R * S)),updP(R,S + 1,SS)
    else st(SID,S + (- rndDelta(R * S))),updP(R,S + 1,SS)
    fi .
eq [updP-owise] : updP(R,S,empty) = empty [owise] .
eq [prefT] : PreferredTraders = 'T2 .

rl [next-rnd] :
    R : SS | TS | OS =>
    R + 1 : updP(R + 1,reSeed(R + 1),SS) | TS | OS .

crl [open-ord] :
    R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,SID,L,PT,SL,
         closed),OS) =>
    R : (st(SID,P),SS) | (tr(TID,C - P),TS) | (ord(OID,TID,SID,L,
         PT,SL,open),OS)
    if P <= L .
crl [close-ord-SL] :
    R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,SID,L,PT,SL,
         open),OS) =>
    R : (st(SID,P),SS) |  (tr(TID,C + L + (- SL)),TS) | OS
    if P <= L - SL .
crl [close-ord-PT] :
    R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,SID,L,PT,SL,
        open),OS) =>
    R : (st(SID,P),SS) | (tr(TID,C + L + PT),TS) | OS
    if P >= L + PT .
```

Figure 4.1: A fragment of the STOCK-EXCHANGE system module.

be traded at a time. This is not a limitation since multiple stocks can be managed by multiple orders.

Basic operations of the stock exchange model (i.e., market time evolution, opening and closure of orders) are implemented via the rules and equations of Figure 4.1. The open-ord rule opens a trader order only if the stock price P falls below or is equal to the order price limit L. When the order is opened, the stock price is subtracted from the trader's capital, thereby updating the capital. Note that, in the set of stocks (`st(SID,P),SS`), the stock `st(SID,P)` is distinguished from all other stocks `SS` in the system.

Similarly, the `close-ord-SL` rule closes an order for the stock `SID` and removes it from the current state when the `SID` stock price `P` falls below or is equal to the `L − SL` stop loss threshold. The trader's capital then increases by the price `P` that the trader gets for the sold stocks. The `close-ord-PT` rule is similar and closes an order when its stock price satisfies the profit target.

Finally, the `next-rnd` rule models the time evolution by simply increasing the round number by one and then automatically updating the stock prices by means of the function `updP`, which randomly increases or decreases the stock prices via the naïve pseudo-random number generator `rndDelta` that is re-seeded at the beginning of each round with the round tick `R + 1`.

Note that the specification of `STOCK-EXCHANGE` given in Figure 4.1 contains two sources of error. First, the function `updP` is flawed because it could generate non-positive stock prices, which are meaningless and should be disallowed. Second, the rule `open-ord` does not check if the available capital of a non-premium trader is enough to cover the order price limit. For instance, for the ordinary Trader `'T`, the following reachability goal[2]

```
(1 : st('S,8) | tr('T,9) | ord('O,'T,'S,12,4,3,closed))
                     =>* R : SS | tr('T,C) | OS
```

which can be solved in Maude via the `search` command, computes (among other solutions) the substitution `{ R / 3, SS / st('S,12), C / - 3, OS / ord('O, 'T, 'S, 12, 4, 3, open) }` that witnesses the existence of an execution trace that starts from the specified initial state and ends in a final state with a faulty, negative capital `C = - 3`.

4.1.2 An Object-oriented Car Rental Store

The second example specifies the `RENT-A-CAR` Full Maude module that models the logic of a (faulty) distributed, object-oriented, online car-rental store that is inspired by a specification in [Clavel et al., 2016]. Similarly to `STOCK-EXCHANGE`, the source code of `RENT-A-CAR` is fully available for consultation in Appendix B.

Here, each state of the system is modeled as a multiset of objects $e_1 \ldots e_n$, where each e_i is:

- a customer that is registered at the store with a certain `credit`,

[2]Given a (possibly) non-ground term `s`, `search` checks whether a reduct of `t` is an instance (modulo the program equations and axioms) of `s` and delivers the corresponding (equational) matcher.

```
class Register | rentals : Nat ,
                 date : Nat .

class Customer | credit : Int,
                 suspended : Bool .

class Car | available : Bool,
            rate : Nat .

class Rental | deposit : Nat,
               dueDate : Nat,
               pickUpDate : Nat,
               customer : Oid,
               car : Oid .

class PreferredCustomer .
class EconomyCar .
class MidSizeCar .
class FullSizeCar .

subclass PreferredCustomer < Customer .
subclasses EconomyCar MidSizeCar FullSizeCar < Car .
```

Figure 4.2: Class and subclass declarations of `RENT-A-CAR`.

- a (rented or available) car,
- a renting contract, or
- the register, which models time elapsing and records the number of active car rentals.

Figure 4.2 depicts the class and subclass declarations of the `RENT-A-CAR` object module. The system considers two kinds of customers: standard customers and preferred customers (who are allowed to rent even if they run out of credit). Basic operations of the store (i.e., rental and return of cars) are implemented via three rewrite rules: `3-day-rental`, `on-date-return`, and `late-return` (see Figure 4.3).

The `3-day-rental` rule enables car rental only if the chosen car is available and, at the time when the contract is signed, the customer makes a `deposit` (that is subtracted from his credit) aimed to cover the estimated charge depending on the daily rental `rate` of the car. Note that the `3-day-rental` rule is flawed because it does not

```
rl [new-day] :
    < RG : Register | date : TODAY > =>
    < RG : Register | date : TODAY + 1 > .

crl [3-day-rental] :
    < U : Customer | credit : CREDIT, suspended : false >
    < C : Car | available : true, rate : RATE >
    < RG : Register | rentals : RNTLS, date : TODAY > =>
    < U : Customer | credit : CREDIT - AMNT >
    < C : Car | available : false >
    < RG : Register | rentals : RNTLS + 1 >
    < qid("R" + string(RNTLS,10)) : Rental | pickUpDate : TODAY,
        dueDate : TODAY + 3, car : C, deposit : AMNT,
        customer : U, rate : RATE >
    if AMNT := 3 * RATE .

crl [on-date-return] :
    < U : Customer | credit : CREDIT >
    < C : Car | rate : RATE >
    < R : Rental | customer : U, car : C, pickUpDate : PDATE,
        dueDate : DDATE, deposit : DPST >
    < RG : Register | date : TODAY > =>
    < U : Customer | credit : (CREDIT + DPST) - AMNT >
    < C : Car | available : true > < RG : Register | >
    if (TODAY <= DDATE) / AMNT := RATE * (TODAY - PDATE) .

crl [late-return] :
    < U : Customer | credit : CREDIT >
    < C : Car | rate : RATE >
    < R : Rental | customer : U, car : C, pickUpDate : PDATE,
        dueDate : DDATE, deposit : DPST >
    < RG : Register | date : TODAY > =>
    updateSuspension(< U : Customer | credit : (CREDIT - AMNT) +
        DPST >)
    < C : Car | available : true > < RG : Register | >
    if DDATE < TODAY / AMNT := RATE * (DDATE - PDATE) +
        (120 * RATE * (TODAY - DDATE)) quo 100 .
```

Figure 4.3: Concurrent rules of the RENT-A-CAR object module.

check whether the current credit of the customer is sufficient to cover the requested deposit, which could lead to erroneous system behaviors.

When a rented car is returned before the due date, the `on-date-return` rule is applied. In this case, the customer is reimbursed for the payment of the initial deposit and is only charged for the number of days he used the car. If the car is returned past the due date, the `late-return` rule is instead applied and the customer is charged an additional sanction that amounts to 20% of the established fee (for each day past the due date).

```
op updateSuspension : Object -> Object .

ceq [suspend] : updateSuspension(
    < U : Customer | credit : CREDIT , suspended : false >) =
    < U : Customer | credit : CREDIT , suspended : true >
    if (CREDIT < 0) .

eq [maintainSuspension] : updateSuspension(
    < U : Customer | suspended : B >) =
    < U : Customer | suspended : B > [owise] .
```

Figure 4.4: Equations modeling customer suspension.

Non-preferred customers are suspended when they reach a negative `credit`, i.e., if they have a debt. Defaulter (non-preferred) customer suspension is modeled by the equations `suspend` and `maintainSuspension` (see Figure 4.4), which applies when the defaulter customer repeats infringement while already suspended. Note that `late-return` rightly admits negative credit and deals with the issue by triggering the function `updateStatus` that suspends the debtor customers who are non-preferred. However, the equations for modeling suspension are erroneous because they cause preferred customers to be suspended as well, which is not what is intended.

4.2 Instrumented Computations

Instrumentation is the ability to monitor the running of a process or a product's performance and to diagnose errors. Maude's computations can be expanded into an *instrumented* computation by explicitly mimicking each application of the matching modulo B algorithm that is used in rewrite and equational simplification steps. In our framework, this is done by means of the specific application of a bogus equational axiom, which is oriented from left to right and then applied as a rewrite rule in the standard way. This is beacuse, typically hidden inside the B-matching algorithms, some pertinent term transformations allow terms that contain operators obeying equational axioms to be rewritten into supportive B-normal forms that facilitate the matching modulo B. In the case of AC-theories, these transformations allow terms to be reordered and correctly parenthesized in order to enable subsequent rewrite steps. Basically, this is achieved by producing a single, auxiliary representative of their AC congruence class (i.e., the AC-normal form). An AC-normal form is typically generated by replacing nested occurrences of the same AC operator by a flattened argument list under a variadic symbol, sorting these arguments under some linear ordering and combining equal arguments using multiplicity superscripts [Eker, 2003]. For example, the congruence class containing $\mathsf{f}(\mathsf{f}(\alpha, \mathsf{f}(\beta, \alpha)), \mathsf{f}(\mathsf{f}(\gamma, \beta), \beta))$ where f is an AC symbol and subterms α, β, and γ belong to alien theories might be represented by $\mathsf{f}^*(\alpha^2, \beta^3, \gamma)$, where f^* is a variadic symbol that replaces nested occurrences of f. A more formal account of this transformation is given in [Eker, 1995].

As for purely associative theories, an A-normal form can be achieved by just flattening nested function symbol occurrences without sorting the arguments. This case has practical importance because it corresponds to lists. C-normal forms are just obtained by properly ordering the arguments of a commutative binary operator. Finally, for function symbols that satisfy the unit axiom U, the unity element of U is not included in the U-normal form, and variables under a U symbol can always be assigned the unity element through U-matching [Eker, 1995].

Then, rewriting modulo B in Maude proceeds by using the special form of matching called B-matching on the internal representation of terms as B-normal forms, where B may contain, among others, any combination of associativity, commutativity, and unity axioms for different binary operators. Moreover, at each Maude step, the resulting term is shown in B-normal form (without multiplicity superscripts).

In the proposed framework, B-matching is simulated by means of specific "fake" axioms that mimick the B-matching transformations of terms that occur internally in the Maude rewrite engine. This allows these transformations to be unhidden and explicitly revealed in the output trace. This artifice is only a means to reveal the term transformations of subterms that are forced by the step so that any position can be properly traced across rewrite steps.

Example 3. *Consider a binary AC operator* f *together with a simple, standard lexicographic ordering over constant symbols. Also consider the term* $\mathsf{t} = \mathsf{f}(\mathsf{b}, \mathsf{f}(\mathsf{f}(\mathsf{b}, \mathsf{a}), \mathsf{c}))$*. Then,* t *matches modulo AC the left-hand side of the rule*

$$[\mathsf{r}] : \mathsf{f}(\mathsf{f}(\mathsf{X}, \mathsf{Y}), \mathsf{f}(\mathsf{Z}, \mathsf{X})) \Rightarrow \mathsf{X}$$

with AC-matching substitutions $\{\, \mathsf{X} \,/\, \mathsf{b}, \mathsf{Y} \,/\, \mathsf{a}, \mathsf{Z} \,/\, \mathsf{c} \,\}$ *and* $\{\, \mathsf{X} \,/\, \mathsf{b}, \mathsf{Y} \,/\, \mathsf{c}, \mathsf{Z} \,/\, \mathsf{a} \}$*. For the first solution, this is mimicked by the transformation sequence*

$$\mathsf{f}(\mathsf{b}, \mathsf{f}(\mathsf{f}(\mathsf{b}, \mathsf{a}), \mathsf{c})) \xrightarrow{\mathtt{toACnf}} \mathsf{f}^*(\mathsf{a}, \mathsf{b}^2, \mathsf{c}) \xrightarrow{\mathtt{fromACnf}} \mathsf{f}(\mathsf{f}(\mathsf{b}, \mathsf{a}), \mathsf{f}(\mathsf{c}, \mathsf{b}))$$

where:

(i) *the first step corresponds to a term transformation that obtains the AC-normal form* $f^*(a, b^2, c)$*, and*

(ii) *the second step corresponds to the inverse, an unflattening transformation that delivers the* AC*-equivalent term* $f(f(b, a), f(c, b))$ *that syntactically matches the left-hand side of* r *with substitution* $\{ X / b, Y / a, Z / c \}$.

Note that an alternative unflattening transformation is possible:

$$f^*(a, b^2, c) \xrightarrow{\text{fromACnf}} f(f(b, c), f(a, b))$$

which actually delivers the second AC-matcher $\{ X / b, Y / c, Z / a \}$. *When several* B*-matchers exist, we only consider those that are effectively computed by means of the Maude internal rewriting strategy.*

Roughly speaking, rewriting modulo B proceeds by using the standard form of B-matching on B-normal forms supported by Maude, where the left-hand sides of the rules are always normalized and the right-hand sides are (partially or totally) normalized when convenient (typically, when the unity element needs to be removed).

Example 4. *Consider two binary AC operators* f *and* g *and the rules*

$$[r_1] : f(c, b, a) \Rightarrow g(c, b, a)$$

$$[r_2] : f(c, f(b, a)) \Rightarrow g(c, g(b, a))$$

whose left-hand (resp. right-hand) sides are pairwise equivalent modulo B.

When the specification that contains them is loaded, the two rules are respectively normalized by Maude into the B*-equivalent rules*

$$[r_1'] : f(a, b, c) \Rightarrow g(a, b, c)$$

$$[r_2'] : f(a, b, c) \Rightarrow g(c, g(a, b))$$

Note that the left-hand side $f(c, b, a)$ *of* r_1 *is reordered as* $f(a, b, c)$ *in* r_1'*, whereas the left-hand side* $f(c, f(b, a))$ *of* r_2 *is not only reordered but also flattened as* $f(a, b, c)$ *in* r_2'.

As for the right-hand sides of the rules, the right-hand side $g(c, b, a)$ *of* r_1 *is reordered as* $g(a, b, c)$ *in* r_1' *whereas the right-hand side of* r_2 *is not flattened in* r_2' *and only the subterm at position 2 (i.e.,* $g(b, a)$*) is reordered; hence, the whole term in the right-hand side of* r_2 *is neither ordered nor flattened in* r_2'.

```
function expand(s, R)
1. $\mathcal{A} = \emptyset$
2. for each $\mathcal{M} \in m\mathcal{S}(s)$
3.   $\mathcal{A} = \mathcal{A} \cup instrument(\mathcal{M})$
4. end for
5. return $\mathcal{A}$
end
```

Figure 4.5: The one-step *expand* function.

In the sequel, when no confusion can arise, we refer to a given program's rule and its corresponding, internally normalized version by using the same label. Therefore, any given instrumented computation consists of a sequence of conditional rewrite steps using the conditional equations ($\rightarrow_\Delta$), conditional rewrite rules ($\rightarrow_R$), equational axioms, and (internal) B-matching transformations ($\rightarrow_B$). More precisely, each rewrite step $s \overset{r,\sigma,w}{\rightarrow}_{R,B} t$ (resp., $s \overset{e,\sigma,w}{\rightarrow}_{\Delta,B} t$) is broken down into a rewrite sequence $s \rightarrow_B^* s' \overset{r,\sigma,w}{\rightarrow}_{R,\emptyset} t' \rightarrow_B^* t$ (resp., $s \rightarrow_B^* s' \overset{e,\sigma,w}{\rightarrow}_{\Delta,\emptyset} t' \rightarrow_B^* t$), where $s' =_B s$ and s' syntactically matches the (normalized) left-hand side of the equation e or rule r that is applied in the considered rewrite step. We define the rewrite relation $\rightarrow_K$ as $\rightarrow_R \cup \rightarrow_\Delta \cup \rightarrow_B$. By *instrument*$(\mathcal{C})$, we denote a function that takes a computation $\mathcal{C}$ and delivers its instrumented counterpart. Nevertheless, in order to improve readability, we omit B-matching transformations and the evaluation of built-in operators when displaying Maude steps (unless explicitly stated otherwise). This is consistent with the strategy adopted by Maude for the case of B-matching transformations, and it is the default option in the ABETS tool (see Chapter 7).

In the following section, an inspection technique is formalized that incrementally builds instrumented computation trees.

4.3 The Exploration Technique

Instrumented computation trees can be easily constructed incrementally by expanding tree nodes (i.e., terms) starting from the root node (i.e., the initial term). Formally, given the term s, the expansion of s in the rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ is defined by the function *expand*$(s, \mathcal{R})$ (see Figure 4.5), which unfolds the term by deploying all the possible instrumented Maude computation steps stemming from s, which is given by $m\mathcal{S}(s)$. In other words, for each Maude step $\mathcal{M} = s \rightarrow_{\Delta,B}^* s{\downarrow_{\Delta,B}} \rightarrow_{R,B} t \rightarrow_{\Delta,B}^* t{\downarrow_{\Delta,B}}$, we first compute its instrumented version and then add the result to the output set $\mathcal{A}$.

```
**function** $explore(s_0, \mathcal{R})$
1. $\mathcal{T}^+_\mathcal{R}(s_0) = s_0$
2. **while**$((s = pickLeaf(\mathcal{T}^+_\mathcal{R}(s_0)) \neq \textbf{EoE})$ **do**
3.   $\mathcal{T}^+_\mathcal{R}(s_0) = addPaths(\mathcal{T}^+_\mathcal{R}(s_0), expand(s, \mathcal{R}))$
4. **end while**
5. **return** $\mathcal{T}^+_\mathcal{R}(s_0)$
**end**
```

Figure 4.6: The *explore* function.

The overall construction methodology for instrumented computation trees is specified by the function *explore*, defined in Figure 4.6. Given a rewrite theory $\mathcal{R}$ and the initial term s_0, the function *explore* essentially formalizes an interactive procedure that is driven by the user starting from an elemental tree fragment, which only consists of the root node s_0. The instrumented computation tree $\mathcal{T}^+_\mathcal{R}(s_0)$ is built by choosing, at each loop iteration of the algorithm, the tree leaf that represents the term to be expanded by means of the auxiliary function $pickLeaf(\mathcal{T}^+_\mathcal{R}(s_0))$, which allows the user to freely select a leaf node from the frontier of the current tree $\mathcal{T}^+_\mathcal{R}(s_0)$. Then, $\mathcal{T}^+_\mathcal{R}(s_0)$ is augmented by calling $addPaths(\mathcal{T}^+_\mathcal{R}(s_0), expand(s, \mathcal{R}))$. This function call adds all the instrumented computations that correspond to the Maude steps that originate from the term s. The special value **EoE** (End of Exploration) is used to terminate the inspection process: when the function $pickLeaf(\mathcal{T}^+_\mathcal{R}(s_0))$ is equal to **EoE**, no term to be expanded is selected and the exploration terminates delivering (a fragment of) the computation tree $\mathcal{T}^+_\mathcal{R}(s_0)$.

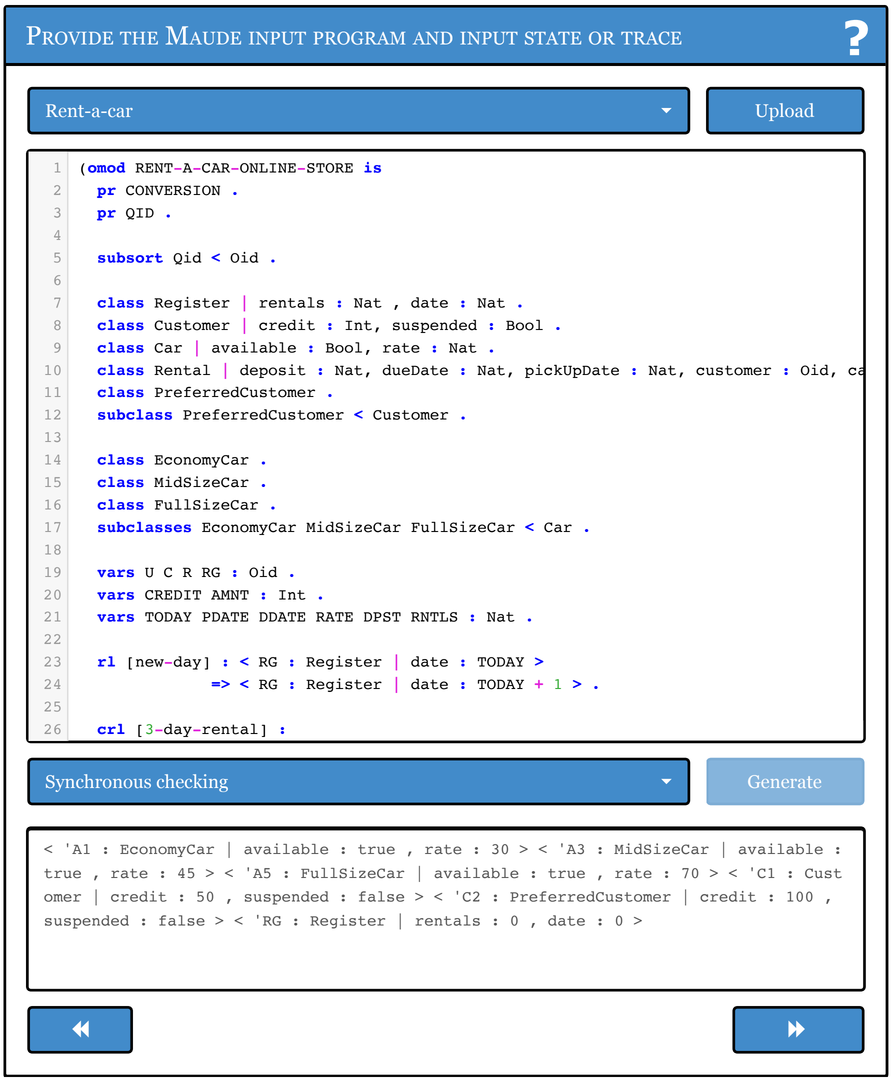
4.4 An Animated Debugging Session

Let us illustrate the exploration methodology proposed in this section by reproducing a debugging session with the analysis and exploration tool ANIMA developed in [Alpuente et al., 2015a].

Consider the buggy, object-oriented Full Maude program `RENT-A-CAR` of Section 4.1.2, which models a car renting online store. Recall that the program behaves in two unintended ways, namely (i) customers can rent cars without enough credit to cover the requested deposit and (ii) preferred customers with negative credit get erroneously suspended. Also, consider the following input state:

```
s0 = < 'A1 : EconomyCar | available : true , rate : 30 >
     < 'A3 : MidSizeCar | available : true , rate : 45 >
     < 'A5 : FullSizeCar | available : true , rate : 70 >
< 'C1 : Customer | credit : 50 , suspended : false >
< 'C2 : PreferredCustomer | credit : 100 , suspended : false >
< 'RG : Register | rentals : 0 , date : 0 >
```

that describes a car rental system with three car object identifiers `'A1`, `'A3`, and 'A5, a non-preferred customer `'C1`, a preferred customer `'C2`, and a register `'RG`.



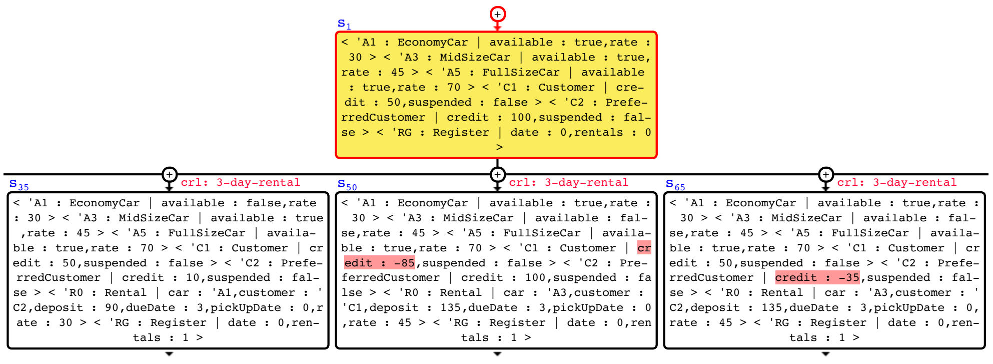
```
(omod RENT-A-CAR-ONLINE-STORE is
  pr CONVERSION .
  pr QID .

  subsort Qid < Oid .

  class Register | rentals : Nat , date : Nat .
  class Customer | credit : Int, suspended : Bool .
  class Car | available : Bool, rate : Nat .
  class Rental | deposit : Nat, dueDate : Nat, pickUpDate : Nat, customer : Oid, ca
  class PreferredCustomer .
  subclass PreferredCustomer < Customer .

  class EconomyCar .
  class MidSizeCar .
  class FullSizeCar .
  subclasses EconomyCar MidSizeCar FullSizeCar < Car .

  vars U C R RG : Oid .
  vars CREDIT AMNT : Int .
  vars TODAY PDATE DDATE RATE DPST RNTLS : Nat .

  rl [new-day] : < RG : Register | date : TODAY >
              => < RG : Register | date : TODAY + 1 > .

  crl [3-day-rental] :
```

```
< 'A1 : EconomyCar | available : true , rate : 30 > < 'A3 : MidSizeCar | available :
true , rate : 45 > < 'A5 : FullSizeCar | available : true , rate : 70 > < 'C1 : Cust
omer | credit : 50 , suspended : false > < 'C2 : PreferredCustomer | credit : 100 ,
suspended : false > < 'RG : Register | rentals : 0 , date : 0 >
```

Figure 4.7: Input Phase.

To start debugging the program, we load the source code of `RENT-A-CAR` and the considered input term in the ANIMA tool [Alpuente et al., 2015a], which is an interactive, parametric exploration tool for Maude and Full Maude programs with (forward and partial) slicing capabilities that implements the techniques described in this chapter. Figure 4.7 illustrates the data input phase that starts the debugging session.

At this moment, the animation of the program proceeds by expanding the first level of the computation tree of RENT-A-CAR. We immediately appreciate that the car rental system is highly concurrent, since the initial state s_0 can be rewritten in seven different states. Specifically, for each customer and each car, a possible rental is modeled. Also, an additional rewrite step that models the pass of time is also computed. By inspecting all seven states, we can detect that five of them include an erroneous customer negative credit, which signals a possible misbehavior. Note that, in the unlikely case when the user knows in advance the wrong data to be searched, this action can be partially automatized by using the query facility included in ANIMA. Specifically, this facility allows users to highlight those fragments of data that match a given query pattern. Figure 4.8 (partially) illustrates the result of querying the computation tree for negative customer credits.



```
< 'A1 : EconomyCar | available : true,rate : 30 > < 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false > < 'C2 : PreferredCustomer | credit : 100,suspended : false > < 'RG : Register | date : 0,rentals : 0 >
```

```
< 'A1 : EconomyCar | available : false,rate : 30 > < 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false > < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'R0 : Rental | car : 'A1,customer : 'C2,deposit : 90,dueDate : 3,pickUpDate : 0,rate : 30 > < 'RG : Register | date : 0,rentals : 1 >
```

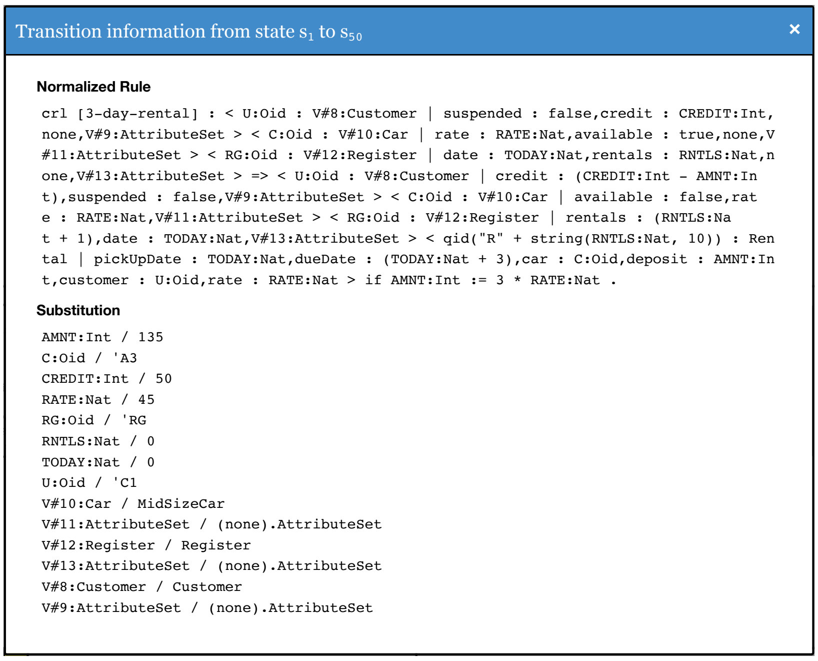
```
< 'A1 : EconomyCar | available : true,rate : 30 > < 'A3 : MidSizeCar | available : false,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : -85,suspended : false > < 'C2 : PreferredCustomer | credit : 100,suspended : false > < 'R0 : Rental | car : 'A3,customer : 'C1,deposit : 135,dueDate : 3,pickUpDate : 0,rate : 45 > < 'RG : Register | date : 0,rentals : 1 >
```

```
< 'A1 : EconomyCar | available : true,rate : 30 > < 'A3 : MidSizeCar | available : false,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false > < 'C2 : PreferredCustomer | credit : -35,suspended : false > < 'R0 : Rental | car : 'A3,customer : 'C2,deposit : 135,dueDate : 3,pickUpDate : 0,rate : 45 > < 'RG : Register | date : 0,rentals : 1 >
```

Figure 4.8: Computation tree after querying for negative credit (partial view).

Now, we can discover that all five erroneous states are generated by the same rewrite rule, namely `3-day-rental`, whose extremely complex normalized form is depicted in Figure 4.9. By carefully analyzing the rule, we can observe that it does not check whether the customer credit covers the 3-day car rental deposit, hence the erroneous negative credit in the observed states.

Therefore, we just encountered the first program flaw, which, luckily, was discovered relatively fast. The second bug (i.e., the erroneous suspension of preferred customers) requires much more effort to be isolated, since it involves a huge fragment of the computation tree to be built and inspected. Actually, with no prior information about the possible erroneous behavior, by using a breadth-first strategy we should generate and analyze a computation tree fragment of thousands of states, which makes the



Transition information from state s_1 to s_{50} ×

Normalized Rule

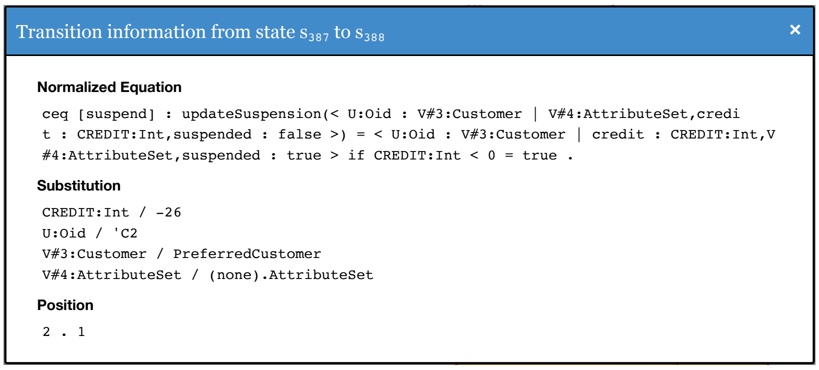
```
crl [3-day-rental] : < U:Oid : V#8:Customer | suspended : false,credit : CREDIT:Int,
none,V#9:AttributeSet > < C:Oid : V#10:Car | rate : RATE:Nat,available : true,none,V
#11:AttributeSet > < RG:Oid : V#12:Register | date : TODAY:Nat,rentals : RNTLS:Nat,n
one,V#13:AttributeSet > => < U:Oid : V#8:Customer | credit : (CREDIT:Int - AMNT:In
t),suspended : false,V#9:AttributeSet > < C:Oid : V#10:Car | available : false,rat
e : RATE:Nat,V#11:AttributeSet > < RG:Oid : V#12:Register | rentals : (RNTLS:Na
t + 1),date : TODAY:Nat,V#13:AttributeSet > < qid("R" + string(RNTLS:Nat, 10)) : Ren
tal | pickUpDate : TODAY:Nat,dueDate : (TODAY:Nat + 3),car : C:Oid,deposit : AMNT:In
t,customer : U:Oid,rate : RATE:Nat > if AMNT:Int := 3 * RATE:Nat .
```

Substitution

```
AMNT:Int / 135
C:Oid / 'A3
CREDIT:Int / 50
RATE:Nat / 45
RG:Oid / 'RG
RNTLS:Nat / 0
TODAY:Nat / 0
U:Oid / 'C1
V#10:Car / MidSizeCar
V#11:AttributeSet / (none).AttributeSet
V#12:Register / Register
V#13:AttributeSet / (none).AttributeSet
V#8:Customer / Customer
V#9:AttributeSet / (none).AttributeSet
```

Figure 4.9: Normalized `3-day-rental` rule.

exploration technique totally unfeasible. Even by considering the minimal fragment of the computation tree that includes a trace that erroneously suspends a preferred customer, we have to manually inspect 38 states (457 instrumented states) to detect the origin of the anomaly in order to discover that the `updateSuspension` function erroneously suspends all kinds of customers whose credit is negative (see Figure 4.10). Figure 4.11 depicts a view of the minimal execution trace that, starting from the input state s_0, ends with the preferred customer `'C2` being erroneously suspended.



Transition information from state s_{387} to s_{388} ×

Normalized Equation

```
ceq [suspend] : updateSuspension(< U:Oid : V#3:Customer | V#4:AttributeSet,credi
t : CREDIT:Int,suspended : false >) = < U:Oid : V#3:Customer | credit : CREDIT:Int,V
#4:AttributeSet,suspended : true > if CREDIT:Int < 0 = true .
```

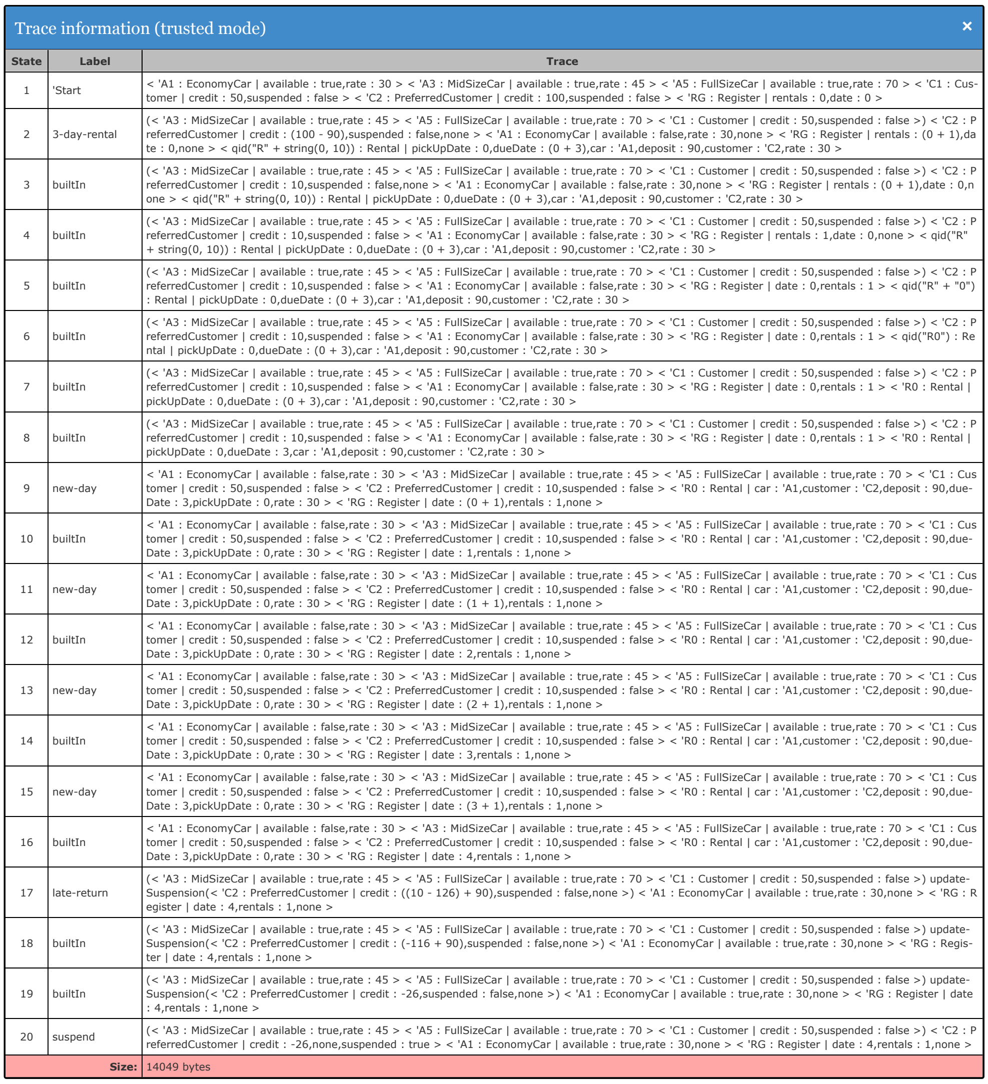
Substitution

```
CREDIT:Int / -26
U:Oid / 'C2
V#3:Customer / PreferredCustomer
V#4:AttributeSet / (none).AttributeSet
```

Position

```
2 . 1
```

Figure 4.10: Normalized `suspend` equation.



Trace information (trusted mode)

State	Label	Trace
1	'Start	< 'A1 : EconomyCar | available : true,rate : 30 > < 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false > < 'C2 : PreferredCustomer | credit : 100,suspended : false > < 'RG : Register | rentals : 0,date : 0 >
2	3-day-rental	(< 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false >) < 'C2 : PreferredCustomer | credit : (100 - 90),suspended : false,none > < 'A1 : EconomyCar | available : false,rate : 30,none > < 'RG : Register | rentals : (0 + 1),date : 0,none > < qid("R" + string(0, 10)) : Rental | pickUpDate : 0,dueDate : (0 + 3),car : 'A1,deposit : 90,customer : 'C2,rate : 30 >
3	builtIn	(< 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false >) < 'C2 : PreferredCustomer | credit : 10,suspended : false,none > < 'A1 : EconomyCar | available : false,rate : 30,none > < 'RG : Register | rentals : (0 + 1),date : 0,none > < qid("R" + string(0, 10)) : Rental | pickUpDate : 0,dueDate : (0 + 3),car : 'A1,deposit : 90,customer : 'C2,rate : 30 >
4	builtIn	(< 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false >) < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'A1 : EconomyCar | available : false,rate : 30 > < 'RG : Register | rentals : 1,date : 0,none > < qid("R" + string(0, 10)) : Rental | pickUpDate : 0,dueDate : (0 + 3),car : 'A1,deposit : 90,customer : 'C2,rate : 30 >
5	builtIn	(< 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false >) < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'A1 : EconomyCar | available : false,rate : 30 > < 'RG : Register | date : 0,rentals : 1 > < qid("R" + "0") : Rental | pickUpDate : 0,dueDate : (0 + 3),car : 'A1,deposit : 90,customer : 'C2,rate : 30 >
6	builtIn	(< 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false >) < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'A1 : EconomyCar | available : false,rate : 30 > < 'RG : Register | date : 0,rentals : 1 > < qid("R0") : Rental | pickUpDate : 0,dueDate : (0 + 3),car : 'A1,deposit : 90,customer : 'C2,rate : 30 >
7	builtIn	(< 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false >) < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'A1 : EconomyCar | available : false,rate : 30 > < 'RG : Register | date : 0,rentals : 1 > < 'R0 : Rental | pickUpDate : 0,dueDate : (0 + 3),car : 'A1,deposit : 90,customer : 'C2,rate : 30 >
8	builtIn	(< 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false >) < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'A1 : EconomyCar | available : false,rate : 30 > < 'RG : Register | date : 0,rentals : 1 > < 'R0 : Rental | pickUpDate : 0,dueDate : 3,car : 'A1,deposit : 90,customer : 'C2,rate : 30 >
9	new-day	< 'A1 : EconomyCar | available : false,rate : 30 > < 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false > < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'R0 : Rental | car : 'A1,customer : 'C2,deposit : 90,dueDate : 3,pickUpDate : 0,rate : 30 > < 'RG : Register | date : (0 + 1),rentals : 1,none >
10	builtIn	< 'A1 : EconomyCar | available : false,rate : 30 > < 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false > < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'R0 : Rental | car : 'A1,customer : 'C2,deposit : 90,dueDate : 3,pickUpDate : 0,rate : 30 > < 'RG : Register | date : 1,rentals : 1,none >
11	new-day	< 'A1 : EconomyCar | available : false,rate : 30 > < 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false > < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'R0 : Rental | car : 'A1,customer : 'C2,deposit : 90,dueDate : 3,pickUpDate : 0,rate : 30 > < 'RG : Register | date : (1 + 1),rentals : 1,none >
12	builtIn	< 'A1 : EconomyCar | available : false,rate : 30 > < 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false > < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'R0 : Rental | car : 'A1,customer : 'C2,deposit : 90,dueDate : 3,pickUpDate : 0,rate : 30 > < 'RG : Register | date : 2,rentals : 1,none >
13	new-day	< 'A1 : EconomyCar | available : false,rate : 30 > < 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false > < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'R0 : Rental | car : 'A1,customer : 'C2,deposit : 90,dueDate : 3,pickUpDate : 0,rate : 30 > < 'RG : Register | date : (2 + 1),rentals : 1,none >
14	builtIn	< 'A1 : EconomyCar | available : false,rate : 30 > < 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false > < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'R0 : Rental | car : 'A1,customer : 'C2,deposit : 90,dueDate : 3,pickUpDate : 0,rate : 30 > < 'RG : Register | date : 3,rentals : 1,none >
15	new-day	< 'A1 : EconomyCar | available : false,rate : 30 > < 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false > < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'R0 : Rental | car : 'A1,customer : 'C2,deposit : 90,dueDate : 3,pickUpDate : 0,rate : 30 > < 'RG : Register | date : (3 + 1),rentals : 1,none >
16	builtIn	< 'A1 : EconomyCar | available : false,rate : 30 > < 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false > < 'C2 : PreferredCustomer | credit : 10,suspended : false > < 'R0 : Rental | car : 'A1,customer : 'C2,deposit : 90,dueDate : 3,pickUpDate : 0,rate : 30 > < 'RG : Register | date : 4,rentals : 1,none >
17	late-return	(< 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false >) updateSuspension(< 'C2 : PreferredCustomer | credit : ((10 - 126) + 90),suspended : false,none >) < 'A1 : EconomyCar | available : true,rate : 30,none > < 'RG : Register | date : 4,rentals : 1,none >
18	builtIn	(< 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false >) updateSuspension(< 'C2 : PreferredCustomer | credit : (-116 + 90),suspended : false,none >) < 'A1 : EconomyCar | available : true,rate : 30,none > < 'RG : Register | date : 4,rentals : 1,none >
19	builtIn	(< 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false >) updateSuspension(< 'C2 : PreferredCustomer | credit : -26,suspended : false,none >) < 'A1 : EconomyCar | available : true,rate : 30,none > < 'RG : Register | date : 4,rentals : 1,none >
20	suspend	(< 'A3 : MidSizeCar | available : true,rate : 45 > < 'A5 : FullSizeCar | available : true,rate : 70 > < 'C1 : Customer | credit : 50,suspended : false >) < 'C2 : PreferredCustomer | credit : -26,none,suspended : true > < 'A1 : EconomyCar | available : true,rate : 30,none > < 'RG : Register | date : 4,rentals : 1,none >
	Size:	14049 bytes

Figure 4.11: Compact View of the faulty trace.

Hence, the faulty program has been successfully debugged by finding the two estimated error sources, namely the `3-day-rental` rule and the `suspend` equation. However, note that during the analysis session we exactly knew which errors to fix and how to locate them. And even so, a number of painful manual inspections were required in order to uncover them. In the following two chapters, an assertion-based, automated trace slicing methodology is formalized that mechanizes and highly improves the debugging experience.

Chapter 5

Runtime Verification of Maude Programs

Assertion checking is one of the most useful automated techniques available for detecting program faults. In runtime assertion checking, assertions are traditionally used to express conditions that should hold at runtime. By finding inconsistencies between specified properties and the program code, runtime assertion checking can prove that the code is incorrect. Moreover, since an assertion failure usually reports an error, the user can often pinpoint the error, direct his attention to the location at which the logical inconsistency is detected, and (hopefully) trace the error back to its origin more easily. Runtime assertion checking can also be useful in finding problems in the specifications themselves, which is important for keeping the specifications accurate and up-to-date. A brief history of the research ideas that have contributed to the assertion capabilities of modern programming languages and development tools can be found in [Clarke and Rosenblum, 2006]. Moreover, the advantages of equipping software with assertions are extensively discussed in [Meyer, 1997].

This chapter presents an assertion-based framework for the runtime verification of rewriting logic specifications. The chapter is organized as follows. Section 5.1 introduces the assertion language and logic of the proposed dynamic verification framework. Section 5.2 formalizes the satisfaction of assertional specifications, while Section 5.3 formalizes the error symptoms of falsified assertions. Finally, Section 5.4 formulates the verification technique that automatically checks Maude programs against assertional specifications.

5.1 The Assertion Language

Assertions are linguistic constructions that formally express properties of a software system. Assertions act as an oracle, giving a pass/fail indication to program runs. Throughout this section, a software system that is specified by a rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ is considered. Without loss of generality, it is assumed that Σ includes at least the sort `State`. Terms of sort `State` are called *system states* (or simply *states*).

In the proposed specification language, assertions are not mere Boolean expressions but truly *executable* formulas that are built on user-defined functions and specialized by means of *state patterns*. The framework supports two kinds of assertions: *functional* assertions and *system* assertions. Functional assertions allow properties to be logically defined on the equational component of the rewrite theory $\mathcal{R}$, while system assertions specify formal constraints on the possibly non-deterministic rule component of $\mathcal{R}$. The benefit of the logic framework being integrated into the Maude specification and analysis environment is that the definition and checking of all asserted properties can be performed in a uniform and familiar setting.

5.1.1 The Assertion Logic

The core of the proposed assertion language is based on order-sorted (membership) predicate logic, where first order formulas are built over the signature Σ of the rewrite theory $\mathcal{R}$ enriched with a set of user-defined Boolean function symbols (predicates). The truth values are given by the formulas `true` and `false`. The usual conjunction (`and`), disjunction (`or`), exclusive or (`xor`), negation (`not`), and implication (`implies`) logic operators are used to express composite properties. Variables in the formulas are not quantified.

Logic formulas can be defined in Maude by means of the predefined functional module `BOOL` [Clavel et al., 2016], which specifies the built-in sort `Bool`, the truth values, the logic operators, and the built-in operators for membership predicates `_:: S` for each sort `S`, and term equality `_==_` and inequality `_=/=_`. The built-in Boolean functions `_==_` and `_=/=_` have a straightforward operational meaning: given an expression `u == v`, then both `u` and `v` are simplified by the equations in the module (which are assumed to be Church-Rosser and terminating) to their canonical forms (modulo the equational axioms) and these canonical forms are compared for equality. If they are equal, the value of `u == v` is `true`; if they are different, it is `false`. The predicate `u =/= v` is just the negation of `u == v`. In the module `BOOL`, valid formulas are reduced to the constant `true`, invalid formulas are reduced to the constant `false`, and all the others are reduced to a canonical form (modulo axioms) consisting of *exclusive or* disjunctions of conjunctions. By default, the `BOOL` module is implicitly imported as a submodule of any other user-defined module.

Predicates that are not specified in `BOOL` are module-dependent and can be equationally defined as total Boolean functions over the system entities (e.g., states, function calls) formalized within $\mathcal{R}$. In the same spirit of Maude's equational theories, where a single result is expected to be delivered for each input term, the framework requires the user to ensure that the evaluation (i.e., the equational simplification) of any property terminates for any possible initial state and that the resulting verdict is unique.

In the proposed framework, basic properties on a given rewrite theory $\mathcal{R}$ are defined by means of a system module PRED($\mathcal{R}$) that imports the (Maude encoding of the) rewrite theory $\mathcal{R}$ and specifies a set P of predicates via user-defined operators that are associated with terminating and Church-Rosser definitions of some total Boolean function. Note that the system module PRED($\mathcal{R}$) must fulfill the same properties as $\mathcal{R}$, that is, its rewrite rules must be coherent with respect to its equations (modulo the equational axioms), and its embedded, extended equational theory, which includes the equational definition of P, must be terminating and Church-Rosser (modulo the equational axioms).

In this scenario, a well-formed formula is any term of sort Bool built using the operators and variables declared in the system module PRED($\mathcal{R}$). Moreover, a formula φ *holds* in $\mathcal{R}$, iff φ can be reduced to true in PRED($\mathcal{R}$) (in symbols, $\mathcal{R} \models \varphi$).

```
mod RENT-A-CAR-PRED is
    pr RENT-A-CAR .

    op isPreferredCustomer : Cid -> Bool .
    eq isPreferredCustomer(PreferredCustomer) = true .
    eq isPreferredCustomer(U:Cid) = false [owise] .

    op isFullSize : Object -> Bool .
    eq isFullSize(< O:Oid : FullSizeCar | available : B:Bool ,
                    rate : RATE:Nat >) = true .
    eq isFullSize(< O:Oid : Car | available : B:Bool ,
                    rate : RATE:Nat >) = false [owise] .
endm
```

Figure 5.1: System properties specified by the RENT-A-CAR-PRED module.

Example 5. *Consider the* RENT-A-CAR *object module of Section 4.1.2 and the new predicate* isFullSize *given in the* RENT-A-CAR-PRED *module of Figure 5.1. Then, we can specify the formula*

```
isFullSize(< O:Oid : FullSizeCar | available : true , rate :
             RATE:Nat >) implies RATE:Nat >= 70
```

which is true for every FullSizeCar *object with an* available *attribute set to* true *and a* rate *attribute greater than or equal to 70.*

5.1.2 System Assertions

System assertions define state invariants that must be satisfied by all system states that match (modulo the equational theory E) a specified state template. Their general syntax is $S\ \{\varphi\}$, where S is a term (called *state template*), φ is a logic formula in conjunctive normal form $\varphi_1 \wedge \ldots \wedge \varphi_n$, and $\mathcal{V}ar(\varphi) \subseteq \mathcal{V}ar(S)$.

Example 6. *Consider the* STOCK-EXCHANGE *example of Section 4.1.1. The following system assertion specifies that the capital of ordinary traders must be non-negative in every system state of the trace:*

```
  R:Nat : SS:Set{Stock} | tr(TID:TraderID,C:Int),TS:Set{Trader} |
                          OS:Set{Order}
      { ordinary(tr(TID:TraderID,C:Int)) implies C:Int >= 0 }
```

where the user-specified predicate ordinary(T) *simply checks whether* T *is a non-premium trader.*

5.1.3 Functional Assertions

Intuitively, functional assertions specify pre- and post-conditions over the equational simplification $t \rightarrow^*_{\Delta} (t{\downarrow_\Delta})$ that heads the rewriting $t \xrightarrow{r}_E t'$ of any term t in the system. Their general form is $I\ \{\varphi_{in}\} \rightarrow O\ \{\varphi_{out}\}$ where $I, O \in \mathcal{T}(\Sigma, \mathcal{V})$, φ_{in} and φ_{out} are well-formed logic formulas, $\mathcal{V}ar(\varphi_{in}) \subseteq \mathcal{V}ar(I)$, and $\mathcal{V}ar(\varphi_{out}) \subseteq \mathcal{V}ar(I) \cup \mathcal{V}ar(O)$.

Example 7. *Consider the* STOCK-EXCHANGE *Maude specification of Section 4.1.1. The functional assertion*

```
      updP(R:Nat,S:Nat,(st(SID:StockID, P:Int),SS:Set{Stock}))
                          { P:Int > 0 }
             -> (st(SID:StockID, P':Int),SS':Set{Stock})
                         { P':Int > 0 }
```

specifies that stock market fluctuations modeled by function updP *should generate positive stock prices provided that the input stock prices are also positive.*

Roughly speaking, functional assertions are implicative formulas between two constrained terms $I\,\{\varphi_{in}\}$ and $O\,\{\varphi_{out}\}$ that specify the general pattern O of the canonical form for any input term t that matches the given template I, while allowing pre- and post-conditions φ_{in}, φ_{out} over the equational simplification to be also declared. This allows users to specify the I/O behavior of the equational simplification of a term t by providing two ingredients:

Input: an input template I that t can match and a pre-condition φ_{in} that t can meet;

Output: an output template O that the canonical form of t has to match and a post-condition φ_{out} that the computed canonical form of t has to meet (whenever the input term t matching I meets φ_{in}).

Note that, while system assertions $S\,\{\varphi\}$ resemble Matching Logic (ML) formulas $\pi \wedge \phi$ (called ML *patterns*), where π is a configuration term and ϕ is a first order logic formula, functional assertions $I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ remind Reachability Logic (RL) formulas $\varphi \Rightarrow \varphi'$, where φ, φ' are ML patterns (for a survey on ML/RL, see [Roşu, 2015]). In contrast to the functional assertions presented in this thesis, which predicate on equational simplifications, RL formulas are evaluated on system computations: the semantics of a RL formula $\varphi \Rightarrow \varphi'$ is that any state satisfying φ transits (in zero or more steps) into a state satisfying φ', while ML formulas are used to express (and reason about) static state properties, similarly to our system assertions. Nevertheless, it is important to recall that the proposed functional assertions are quantifier-free and can be efficiently evaluated by relying on Maude standard infrastructure such as the `metaReduce`, `metaMatch`, and `metaNormalize` meta-operations.

5.1.4 Assertional Specifications

An *assertional specification* $\mathcal{A}$ for a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is thus a set of functional and system assertions for $\mathcal{R}$. $\mathcal{F}(\mathcal{A})$ denotes the set of functional assertions in $\mathcal{A}$, while $\mathcal{S}(\mathcal{A})$ denotes the set of system assertions in $\mathcal{A}$. Moreover, $s \models \mathcal{S}(\mathcal{A})$ (resp. $\mu \models \mathcal{F}(\mathcal{A})$) denotes that s satisfies all assertions in $\mathcal{S}(\mathcal{A})$ (resp. μ satisfies all assertions in $\mathcal{F}(\mathcal{A})$). Figure 5.2 summarizes the proposed assertion language.

In the following section, the notion of satisfaction for functional and system assertions is described in detail.

5.2 Satisfaction of Assertions

This section formalizes the notion of satisfaction for system and functional assertions.

5.2.1 System Assertion Satisfaction

System assertions are checked against states of the system that is specified by $\mathcal{R}$. Roughly speaking, a system assertion $S\,\{\varphi\}$ (also called constrained term in [Rocha

Formulas

$$\varphi ::= \varphi \wedge \varphi \mid \mathit{not}\ \varphi \mid \varphi\ \mathit{and}\ \varphi \mid \varphi\ \mathit{or}\ \varphi \mid$$
$$\varphi\ \mathit{implies}\ \varphi \mid \mathit{true} \mid \mathit{false} \mid t$$

where $t \in \mathcal{T}(\Sigma', \mathcal{V})_{Bool}$ and $\Sigma' \supseteq \Sigma$.

System assertions

$$\mathrm{S}\ \{\varphi\}$$

where $\mathrm{S} \in \mathcal{T}(\Sigma, \mathcal{V})$ and $\mathcal{V}ar(\varphi) \subseteq \mathcal{V}ar(\mathrm{S})$.

Functional assertions

$$\mathrm{I}\ \{\varphi_{in}\} \longrightarrow \mathrm{O}\ \{\varphi_{out}\}$$

where $\mathrm{I}, \mathrm{O} \in \mathcal{T}(\Sigma, \mathcal{V})$, $\mathcal{V}ar(\varphi_{in}) \subseteq \mathcal{V}ar(\mathrm{I})$,
and $\mathcal{V}ar(\varphi_{out}) \subseteq \mathcal{V}ar(\mathrm{I}) \cup \mathcal{V}ar(\mathrm{O})$.

Figure 5.2: Summary of the assertion language.

et al., 2014]) allows users to validate all system states s that match (modulo the equational theory E) the state template S with respect to the formula φ. More formally, the satisfaction of a system assertion in a system state is defined as follows.

Definition 1 (system assertion satisfaction)**.** *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory. Let $S\ \{\varphi\}$ be a system assertion for $\mathcal{R}$ and s be a state in $\mathcal{T}(\Sigma, \mathcal{V})$. Then, $S\ \{\varphi\}$ is* satisfied *in s (in symbols, $s \models S\ \{\varphi\}$) iff for each $w \in \mathcal{P}os(s)$, for each substitution σ if $s|_w =_E S\sigma$ then $\varphi\sigma$ holds in $\mathcal{R}$.*

Note that, if there is no subterm $s|_w$ of s that matches S (modulo E), then trivially $s \models S\ \{\varphi\}$. This implies that $S\ \{\varphi\}$ is *not* satisfied in s (in symbols, $s \not\models S\ \{\varphi\}$) only in the case when there exist w and σ such that $s|_w =_E S\sigma$, and the formula $\varphi\sigma$ does not hold in $\mathcal{R}$.

Example 8. *Consider the* `STOCK-EXCHANGE` *specification of Section 4.1.1 and the system assertion of Example 6. Also consider the system state*

```
s = 1 : st('S1, 4) | tr('T1, 2) | empty
```

which models a stock-exchange system consisting of a single stock, namely `'S1` *with a stock price of* `4`*, and a trader* `'T1` *with a positive capital of* `2`*. Then, the state pattern* S *of the system assertion matches* s *at position* Λ *with substitution*

$$\sigma = \{\ \texttt{R:Nat / 1, SS:Set\{Stock\} / st('S1, 4), TID:TraderID / 'T1,}\ \texttt{C:Int / 2, TS:Set\{Trader\} / empty, OS:Set\{Order\} / empty}\ \}$$

and the formula φ *of the assertion is instantiated as follows*

$$\varphi\sigma = \texttt{ordinary(tr('T1,2)) implies 2 >= 0}$$

which is reduced to `true` *since* `'T1` *is not a premium trader and his/her capital is greater than or equal to* `0`*. Hence, the state* s *satisfies the considered system assertion. However, for a slightly mutated system state* s'*, where the original capital of trader* `'T1` *is changed to* `- 2`*, the formula* $\varphi\sigma'$ *is reduced to* `false`*, which means that* s' *does* not *satisfy the assertion.*

5.2.2 Functional Assertion Satisfaction

The notion of satisfaction for a functional assertion is given with respect to the equational simplification $\mu = t \rightarrow^*_{\Delta,B} t{\downarrow_{\Delta,B}}$ of term t into its canonical form $t{\downarrow_{\Delta,B}}$.

Definition 2 (functional assertion satisfaction)**.** *Let* $\mathcal{R} = (\Sigma, E, R)$ *be a rewrite theory, with* $E = \Delta \cup B$*. Let* $I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ *be a functional assertion for* $\mathcal{R}$*, and* μ *be the equational simplification of the term* t *in* $\mathcal{T}(\Sigma, \mathcal{V})$ *into its canonical form* $t{\downarrow_{\Delta,B}}$ *with respect to* Δ *modulo* B*. Then,* $I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ *is* satisfied *in* μ *(in symbols,* $\mu \models I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$*) iff for each substitution* σ_{in} *such that* $t =_B I\sigma_{in}$*, if* $\varphi_{in}\sigma_{in}$ *holds in* $\mathcal{R}$*, then there exists* σ_{out} *such that* $t{\downarrow_{\Delta,B}} =_B O(\sigma_{in}{\downarrow_{\Delta,B}})\sigma_{out}$ *and* $\varphi_{out}(\sigma_{in}{\downarrow_{\Delta,B}})\sigma_{out}$ *holds in* $\mathcal{R}$*.*

The satisfaction of functional assertions could be equivalently defined on the call term t (rather than on its equational simplification $\mu : t \rightarrow^*_{\Delta,B} t{\downarrow_{\Delta,B}}$) since the normal form $t{\downarrow_{\Delta,B}}$ is uniquely defined in a canonical equational theory. Nonetheless, it is preferable to define the satisfaction with respect to μ since this notion is much closer to the intuitive meaning of functional assertions (whose satisfiability depends on both the input term t and the reduced term $t{\downarrow_{\Delta,B}}$ of μ). Therefore, using μ greatly simplifies the description.

Note that $I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ is (trivially) satisfied in μ when either t does not match I (modulo B), or $t =_B I\sigma_{in}$ and $\varphi_{in}\sigma_{in}$ does not hold in $\mathcal{R}$. Intuitively, a functional error occurs in an equational simplification μ where the computed canonical form fails to match the structure or meet the properties of the output template O. In other words, $\Phi = I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ is *not* satisfied in μ only in the case when there exists an *input* substitution σ_{in} (i.e., a substitution that matches t within the input template I modulo B; in symbols, $t =_B I\sigma_{in}$) such that

- $\varphi_{in}\sigma_{in}$ holds in $\mathcal{R}$;
- $t{\downarrow_{\Delta,B}} \neq_B O(\sigma_{in}{\downarrow_{\Delta,B}})\sigma_{out}$ or $\varphi_{out}(\sigma_{in}{\downarrow_{\Delta,B}})\sigma_{out}$ does not hold in $\mathcal{R}$, for any substitution σ_{out}.

Example 9. *Consider the* `STOCK-EXCHANGE` *specification of Section 4.1.1 and the functional assertion of Example 7. Also consider the simplification trace*

$$\mu = \texttt{updP(1 + 2, reSeed(1 + 2), (st('S1, 4),st('S2, 12)))} \\ \to^{*} \texttt{st('S1, 10),st('S2, - 2)}$$

which leads

$$t = \texttt{updP(1 + 2, reSeed(1 + 2), (st('S1, 4),st('S2, 12)))}$$

to its canonical form. The input state pattern I *of the functional assertion matches* t *with substitution*

$$\sigma_{in} = \{\ \texttt{R:Nat / 1 + 2, S:Nat / reSeed(1 + 2), SID:StockID / 'S1,} \\ \texttt{P:Int / 4, SS:Set\{Stock\} / st('S2,12) \}.}$$

Then, σ_{in} *is simplified and applied to output state pattern* O *of the assertion and results in*

$$O(\sigma_{in}{\downarrow_{\Delta,B}}) = \texttt{(st('S1, P':Int),st('S2,12)).}$$

Therefore, $O(\sigma_{in}{\downarrow_{\Delta,B}})$ *matches* $t{\downarrow_{\Delta,B}}$ *with substitution*

$$\sigma_{out} = \{\ \texttt{P':Int / 10}\}$$

On the other hand, the input formula φ_{in} *is instantiated as follows*

$$\varphi_{in}\sigma_{in} = \texttt{4 > 0}$$

which trivially reduces to `true`*, whereas the instantiation of the output formula* φ_{out} *results in the following concretization of the formula*

$$\varphi_{out}(\sigma_{in}{\downarrow_{\Delta,B}})\sigma_{out} = \texttt{10 > 0}$$

which is also simplified to `true`*. Hence, it might seem that* μ *satisfies the given assertion, but actually, it does not. Note that* t *also matches the input state pattern* I *with substitution*

$$\sigma'_{in} = \{\ \texttt{R:Nat / 1 + 2, S:Nat / reSeed(1 + 2), SID:StockID / 'S2,}\ \texttt{P:Int / 12, SS:Set\{Stock\} / st('S1,4)}\ \}$$

which reduces the instantiated input formula φ_{in} to `true`*, since*

$$\varphi_{in}\sigma'_{in} = \texttt{12 > 0.}$$

Moreover, the output state pattern O is (partially) instantiated as

$$O(\sigma'_{in}{\downarrow_{\Delta,B}}) = \texttt{(st('S2, P':Int),st('S1,4)).}$$

Therefore, $O(\sigma'_{in}{\downarrow_{\Delta,B}})$ matches $t{\downarrow_{\Delta,B}}$ with output substitution

$$\sigma'_{out} = \texttt{\{ P':Int / - 2\}}$$

and the formula φ_{out} of the functional assertion gets evaluated as

$$\varphi_{out}(\sigma'_{in}{\downarrow_{\Delta,B}})\sigma'_{out} = \texttt{- 2 > 0,}$$

which reduces to `false`*. Hence, now we can safely state that μ does not satisfy the given functional assertion, since at least one of the input substitutions causes the output formula not to hold in the considered theory.*

Given the set P of new user-defined predicates and their equational definition Q, it is worth noting that we could split the set of all functions defined in the extended equational theory $E \cup Q$ into two disjoint sets, $U \oplus T$, where U are the untrusted functions of E (those to be debugged) and T is the extension of P with the set of all trusted functions defined in E. Now, by requiring that T includes all functions allowed in admissible functional assertions plus the functions they depend on (which can be easily approximated by analyzing the graph of functional dependencies of the extended theory), we do not even need the canonicity of the whole equational theory $E \cup Q$; we only need the canonicity of the sub-theory that defines the trusted set T.

5.3 Uncovering Error Symptoms

To be able to precisely identify and isolate the subterms responsible for the violation of an assertion is a huge advantage, since it allows provenance techniques to be applied, which faithfully track the origins of the detected errors thus opening the door to possible repairs. Actually, the more accurate the detected errors, the higher automation achieved in the subsequent debugging process. This section formalizes the computation of accurate error symptoms from both system and functional assertions.

5.3.1 System Error Symptoms

When a system state s does not satisfy a system assertion $S\,\{\varphi\}$, the position w in s (i.e., the *bug* position) precisely indicates the subterm of s that matches S and is responsible for the assertion violation. The position w is called then a *system error symptom*.

Definition 3 (system error symptoms)**.** *The set of all system error symptoms for a state s and a system assertion $S\,\{\varphi\}$ is defined as follows:*

$$\xi_{sys}(s, S\,\{\varphi\}) = \{w \mid \exists\sigma.\ s|_w =_E S\sigma, w \in \mathcal{P}os(s),\ \textit{and}\ \mathcal{R} \not\models \varphi\sigma\}.$$

Observe that $\xi_{sys}(s, S\,\{\varphi\}) = \emptyset$, whenever $s \models S\,\{\varphi\}$.

Example 10. *Consider the extended rewrite theory of Example 5 together with the system assertion*

$\Theta =$
```
< O:Oid : C:Cid | credit : B:Int , suspended : S:Bool >
{ not(isPreferredCustomer(C:Cid)) implies B:Int >= 0 }
```

Then, Θ is satisfied in the state

```
< 'A5 : FullSizeCar | available : true , rate : 70 >
< 'C1 : Customer | credit : 50 , suspended : false >
< 'RG : Register | date : 0 , rentals : 0 >
```

but it is not satisfied in

$s_{err} =$
```
< 'A5 : FullSizeCar | available : false , rate : 70 >
< 'C1 : Customer | credit : - 160 , suspended : false >
< 'R0 : Rental | car : 'A5 , customer : 'C1 , deposit : 210,
dueDate : 3 , pickUpDate : 0 , rate : 70 >
< 'RG : Register | date : 0 , rentals : 1 >
```

since non-preferred customer `'C1` *has a negative credit. The computed error symptom is the position 2 that refers to the subterm*

```
< 'C1 : Customer | credit : - 160 , suspended : false >
```

of the anomalous state s_{err}.

5.3.2 Functional Error Symptoms

Unlike system error symptoms, functional error symptoms are a little more complex to uncover, since functional assertions can be violated in two different ways. Specifically, functional error symptoms are specialized in two different modalities, as shown in Definition 4.

Definition 4 (functional error symptoms)**.** *Let* $\mathcal{R} = (\Sigma, E, R)$ *be a rewrite theory, with* $E = \Delta \cup B$*. Let* $\Phi = I\,\{\varphi_{in}\} \to O\,\{\varphi_{out}\}$ *be a functional assertion for* $\mathcal{R}$*. Let* $\mu = t \to^{*}_{\Delta,B} t{\downarrow_{\Delta,B}}$ *be an equational simplification such that* $\mu \not\models \Phi$ *with input substitution* σ_{in}*. Then, a* functional error symptom *for* μ *with respect to* Φ *is any position in* $\mathcal{P}os(t{\downarrow_{\Delta,B}})$ *that belongs to the following set:*

$$\xi_{fun}(\mu,\Phi) = \begin{cases} \{\pi(w) \mid ((g,\sigma_1,\sigma_2),\pi) = \widehat{lgg}_B(t{\downarrow_{\Delta,B}}, O(\sigma_{in}{\downarrow_{\Delta,B}})) \\ \qquad\qquad\qquad\qquad \textit{and } w \in \mathcal{VP}os(g)\} & (1) \\ \qquad\qquad \textit{if } \nexists \sigma_{out} \text{ s.t. } t{\downarrow_{\Delta,B}} =_B O(\sigma_{in}{\downarrow_{\Delta,B}})\sigma_{out} \\ \{\Lambda\} \qquad\quad \textit{if } \forall \sigma_{out} \text{ s.t. } t{\downarrow_{\Delta,B}} =_B O(\sigma_{in}{\downarrow_{\Delta,B}})\sigma_{out}, \\ \qquad\qquad\qquad\qquad \mathcal{R} \not\models \varphi(\sigma_{in}{\downarrow_{\Delta,B}})\sigma_{out} & (2) \end{cases}$$

Roughly speaking, $\xi_{fun}(\mu,\Phi)$ is computed by distinguishing two cases.

Case (1) If no matching substitution σ_{out} exists that allows the canonical form $t{\downarrow_{\Delta,B}}$ to be matched within the instance $O(\sigma_{in}{\downarrow_{\Delta,B}})$ of the output template O by the (normalized) substitution $\sigma_{in}{\downarrow_{\Delta,B}}$, the technique "compares" $t{\downarrow_{\Delta,B}}$ with $O(\sigma_{in}{\downarrow_{\Delta,B}})$ by using a least general generalization algorithm modulo equational theories. More specifically, an arbitrarily-selected least general generalization (g,σ_1,σ_2) (modulo $\Delta \cup B$) between $t{\downarrow_{\Delta,B}}$ and $O(\sigma_{in}{\downarrow_{\Delta,B}})$ is chosen via $\widehat{lgg}_B$, and erroneous subterms of $t{\downarrow_{\Delta,B}}$ are detected by selecting every position $\pi(w) \in \mathcal{P}os(t{\downarrow_{\Delta,B}})$ in correspondence with a position $w \in \mathcal{VP}os(g)$. The intuition behind this method is that variables in g reflect the discrepancies between the computed canonical form and the instantiated output template, and therefore subterms $(t{\downarrow_{\Delta,B}})|_{\pi(w)}$ represent anomalies in $t{\downarrow_{\Delta,B}}$.

Case (2) If for every matcher (modulo B) σ_{out} of the computed canonical form $t{\downarrow_{\Delta,B}}$ in $O(\sigma_{in}{\downarrow_{\Delta,B}})$, the (instantiated) formula $\varphi(\sigma_{in}{\downarrow_{\Delta,B}})\sigma_{out}$ does not hold in $\mathcal{R}$, then $t{\downarrow_{\Delta,B}}$ does not meet the property φ and its root position (which identifies the whole erroneous term) is signalled as a functional error symptom. Note that, in this case, the detection of the error source could be only roughly approximated, since the whole computed canonical form is considered faulty, even though only some parts could be responsible for the error.

Example 11. *Consider again the extended rewrite theory of Example 5. Then, the functional assertion*

$$\begin{aligned}\Phi = \;& \texttt{updateSuspension(< U:Oid : PreferredCustomer | credit : B:Int,}\\ & \texttt{suspended : false >)}\{\, \texttt{B:Int < 0} \,\}\\ \rightarrow \;& \texttt{< U:Oid : PreferredCustomer | credit : B:Int,}\\ & \texttt{suspended : false >}\{\, \texttt{true} \,\}\end{aligned}$$

states that, for preferred customers, the suspended *flag (and other customer attributes) remain unchanged after* updateSuspension *is invoked. Roughly speaking, preferred customers are never suspended, even if they were slow payers. Thus,* Φ *is not satisfied in the following equational simplification*

```
updateSuspension(< 'C1 : PreferredCustomer | credit : - 25,
suspended : false >)
```

$\xrightarrow{\texttt{suspend}}$

```
< 'C1 : PreferredCustomer | credit : - 25, suspended : true >
```

with input substitution $\sigma_{in} = \{\, \texttt{U:Oid / 'C1, B:Int / - 25} \,\}$*. The violation of* Φ *corresponds to case (1) of Definition 4, since the computed canonical form for the* updateSuspension *function call does not match the instantiation of the output template* Φ *with* $\sigma_{in}{\downarrow_{\Delta,B}}$ *(which is equal to* σ_{in} *in this case). Hence, the technique computes the only (actually syntactical) least general generalization*

$$\begin{aligned}&\widehat{lgg}_{B}(\texttt{< 'C1 : PreferredCustomer | credit : - 25, suspended : true >,}\\ &\texttt{< U:Oid : PreferredCustomer | credit : B:Int, suspended : false >}\\ &(\sigma_{in}{\downarrow_{\Delta,B}})) \texttt{ = ((< 'C1 : PreferredCustomer | credit : - 25, suspended :}\\ &\texttt{X:Bool >,\{ X:Bool / true \},\{ X:Bool / false \}),\{3.2.1} \mapsto \texttt{3.2.1\})}\end{aligned}$$

where $\xi_{fun}(\mu, \Phi) = \{3.2.1\}$ *is the set of functional error symptoms that pinpoint the anomalous* suspended *flag value in* 'C1*'s data structure, that is,*

$$\begin{aligned}&\texttt{< 'C1 : PreferredCustomer | credit : - 25,}\\ &\texttt{suspended : true >}|_{3.2.1} = \texttt{true.}\end{aligned}$$

Now, consider this slight mutation of the assertion Φ

$$\begin{aligned}\Phi' = \;& \texttt{updateSuspension(< U:Oid : PreferredCustomer | credit :}\\ & \texttt{B:Int , suspended : S:Bool >)}\{\, \texttt{B:Int < 0} \,\}\\ \rightarrow \;& \texttt{< U:Oid : PreferredCustomer | credit : B:Int ,}\\ & \texttt{suspended : S':Bool >}\{\, \texttt{S:Bool == S':Bool} \,\}\end{aligned}$$

whose post-condition explicitly states that updateSuspension *calls cannot change the value of the* suspended *flag. Also* Φ' *is not satisfied in the equational simplification above, but, in this case, the reason of the violation stands in the refutation of the* *(instantiated) post-condition, which corresponds to case (2) of Definition 4. Therefore, our methodology delivers* $\xi_{fun}(\mu, \Phi') = \{\Lambda\}$*, thereby providing a less precise error detection analysis that marks the whole computed canonical form*

```
< 'C1 : PreferredCustomer | credit : - 25 , suspended : true >
```

as incorrect.

It is worth noting that the use of $\widehat{lgg}_B$ is generally preferable to the adoption of a pure syntactic *lgg* algorithm since it minimizes the number of variables in g and, hence, the points of discrepancy between $t\downarrow_{\Delta,B}$ and $O(\sigma_{in}\downarrow_{\Delta,B})$, which facilitates isolating erroneous information. Let us see an example.

Example 12. *Let us consider the equational simplification* $\mathtt{f}(0,0) \to^{+}_{\Delta,B} \mathtt{c}(1,3)$ *with respect to an equational theory* $(\Sigma, \Delta \cup B)$ *in which the operator* c *is declared commutative. Let* $\Phi = \mathtt{f(X,Y)\ \{\ true\ \}} \to \mathtt{c(Z,1)\ \{\ even(Z)\ \}}$ *be a functional assertion, where predicate* even(Z) *checks whether* Z *is an even number.*

Then, $(\mathtt{f}(0,0), \mathtt{c}(1,3)) \not\models \Phi$ *(with input substitution* $\sigma_{in} = \{X/0, Y/0\}$*), since variable* Z *in the output template* c(Z,1) *is bound to* 3 *and* even(3) *is false. Moreover,* $\widehat{lgg}_B(\mathtt{c}(1,3), \mathtt{c}(Z,1))$ *returns a pair* $((g, \sigma_1, \sigma_2), \pi)$ *such that* g *contains the minimum number of variables. For instance,*

$$\widehat{lgg}_B(\mathtt{c}(1,3), \mathtt{c}(Z,1)) = ((\mathtt{c}(Z,1), \{Z/3\}, \{\}), \{1 \mapsto 2\})$$

and $\xi_{fun}(\mu, \Phi) = \{2\}$*, which precisely detects that the term* $\mathtt{c}(1,3)|_2 = 3$ *is what causes the violation of* Φ*.*

By contrast, the computation of a purely syntactic least general generalization would have delivered the more general result $(\mathtt{c}(Z,W), \{Z/1, W/3\}, \{W/1\})$ *and the larger functional error symptom set* $\{1,2\}$ *(which represents the positions of both arguments of the canonical form* c(1,3)*), thereby hindering the isolation of the erroneous subterm of* c(1,3)*.*

5.4 Dynamic Assertion-Checking

Let us first extend the notion of satisfaction of the functional assertions to state equational simplifications (i.e., equational simplifications that reduce a state into its canonical form), where the state may contain an arbitrary number of function calls that might eventually be simplified. For this purpose, we introduce the following auxiliary definitions. Given $\mathcal{R} = (\Sigma, E, R)$, with $E = \Delta \cup B$, the term t is an equational redex in $\mathcal{R}$ if there is $(\lambda = \rho\ \mathit{if}\ C) \in \Delta$ and substitution σ such that $t =_B \lambda\sigma$. Given $\mathcal{R}$ and a system state s in $\mathcal{T}(\Sigma, \mathcal{V})$, $\mathrm{Top}(s)$ is the set of minimal positions $w \in \mathcal{P}os(s)$ such that $s|_w$ is an equational redex in $\mathcal{R}$. Formally,

$$Top(s) = \{w \in \mathcal{P}os(s) \mid s|_w \text{ is an equational redex and } \\ \nexists w' \leq w \text{ such that } s|_{w'} \text{ is an equational redex}\}.$$

Roughly speaking, $Top(s)$ selects all the positions in $\mathcal{P}os(s)$ that identify those outermost subterms of s to be equationally simplified into their canonical form in order to compute $s{\downarrow_{\Delta,B}}$. In other words, given the equational simplification of the state s, $\mathcal{S} : s \to^{+}_{\Delta,B} s{\downarrow_{\Delta,B}}$, each subterm $s|_w$, with $w \in Top(s)$, is reduced to $(s|_w{\downarrow_{\Delta,B}})$ in $\mathcal{S}$. This allows functional assertions to be effectively checked over each equational simplification $s|_w \to^{+}_{\Delta,B} (s|_w{\downarrow_{\Delta,B}})$ such that $w \in Top(s)$.

Definition 5 (extended functional assertion satisfaction)**.** *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, with $E = \Delta \cup B$, and let s be a system state in $\mathcal{T}(\Sigma, \mathcal{V})$ such that $Top(s) \neq \{\Lambda\}$. Let $s \to^{+}_{\Delta,B} s{\downarrow_{\Delta,B}}$ be an equational simplification for the state s in $\mathcal{T}(\Sigma, \mathcal{V})$. Let $\mathcal{A}$ be an assertional specification for $\mathcal{R}$. We say that $\mathcal{F}(\mathcal{A})$ is* satisfied *in $s \to^{+}_{\Delta,B} s{\downarrow_{\Delta,B}}$ (in symbols, $s \to^{+}_{\Delta,B} s{\downarrow_{(\Delta,B)}} \models \mathcal{F}(\mathcal{A})$), iff for each $w \in Top(s)$, $s|_w \to^{+}_{\Delta,B} (s{\downarrow_{\Delta,B}})|_w \models \mathcal{F}(\mathcal{A})$.*

System and functional error symptoms (whose definitions have been given in Section 5.3 for a single system/functional assertion) can be naturally extended to assertional specifications in the following way.

Definition 6 (state error symptoms)**.** *Let $\mathcal{R} = (\Sigma, E, R)$, with $E = \Delta \cup B$, be a rewrite theory. Let $\mathcal{A}$ be an assertional specification for $\mathcal{R}$. Let s be a state in $\mathcal{T}(\Sigma, \mathcal{V})$. Then,*

$$\xi_{sys}(s, \mathcal{A}) = \bigcup_{\Theta \in \mathcal{S}(\mathcal{A})} \xi_{sys}(s, \Theta)$$

$$\xi_{fun}(s \to^{+}_{\Delta,B} s{\downarrow_{\Delta,B}}, \mathcal{A}) = \bigcup_{\substack{\Phi \in \mathcal{F}(\mathcal{A}),\\ w \in Top(s)}} \{(s|_w \to^{+}_{\Delta,B} (s{\downarrow_{\Delta,B}})|_w, \xi_{fun}(s|_w \to (s{\downarrow_{\Delta,B}})|_w, \Phi))\}$$

Example 13. *Consider the rewrite theory $\mathcal{R}$ of Example 5 together with the assertional specification $\mathcal{A}$ composed of the system assertion*

```
Θ =  < O:Oid : C:Cid | credit : B:Int , suspended : S:Bool >
     { not(isPreferredCustomer(C:Cid)) implies B:Int >= 0 }
```

and the functional assertion

```
Φ =  updateSuspension(< U:Oid : PreferredCustomer | credit : B:Int
     , suspended : false >) { B:Int < 0 }
 →   < U:Oid : PreferredCustomer | credit : B:Int , suspended :
     false > { true }
```

Let $\mu_{\mathtt{rent}}$ *be the state equational simplification that originates from*

```
s =  < 'A1 : EconomyCar | available : true , rate : 30 >
     < 'C1 : Customer | credit : - 160 , suspended : false >
     updateSuspension(< 'C2 : PreferredCustomer | credit : - 120,
     suspended : false >)
     < 'RG : Register | date : 0 , rentals : 1 >
```

and ends into the canonical form

```
s↓Δ,B=  < 'A1 : EconomyCar | available : true , rate : 30 >
        < 'C1 : Customer | credit : - 160 , suspended : false >
        < 'C2 : PreferredCustomer | credit : - 120 ,
        suspended : true >
        < 'RG : Register | date : 0 , rentals : 1 >
```

Note that $\mu_{\mathtt{rent}}$ *includes the following equational simplification*

$$\mu_{'\mathtt{C2}} = \mathtt{updateSuspension(< 'C2 : PreferredCustomer | credit : - 120 , suspended : false >)} \rightarrow^{+}_{\Delta,B} \mathtt{< 'C2 : PreferredCustomer | credit : - 120 , suspended : true >}$$

for the outermost equational redex of s *that is rooted at position* $3 \in Top(\mathsf{s})$.

Then,

$$\xi_{sys}(\mathsf{s}, \mathcal{A}) = \{2\}$$

which signals the system error symptom associated with the negative credit of non-preferred customer 'C1.

Moreover,

$$\xi_{fun}(\mu_{\mathtt{rent}}, \mathcal{A}) = \{(\mu_{'\mathtt{C2}}, \{3.2.1\})\}$$

since Φ *is not satisfied in* $\mu_{\mathtt{rent}}$ *(and hence in* $\mu_{'\mathtt{C2}}$*). The computed functional error symptom allows us to isolate the anomalous* suspended *flag value in* 'C2*'s data structure, that is,*

$$\mathtt{< 'C2 : PreferredCustomer | credit : - 120, suspended : true >}|_{3.2.1} = \mathtt{true}.$$

The notion of satisfaction for an assertional specification in a given computation is then formalized as follows.

Definition 7 (satisfaction of an assertional specification)**.** *Let* $\mathcal{R} = (\Sigma, E, R)$, *with* $E = \Delta \cup B$, *be a rewrite theory and* $\mathcal{C}$ *be a computation in* $\mathcal{R}$. *Let* $\mathcal{A}$ *be an assertional specification for* $\mathcal{R}$. *Then, the specification* $\mathcal{A}$ *is* satisfied *in* $\mathcal{C}$ *(in symbols* $\mathcal{C} \models \mathcal{A}$*) iff*

- *for each state* s *in* $\mathcal{C}$ *that is a canonical form with respect to* Δ *modulo* B, $s \models \mathcal{S}(\mathcal{A})$*;*
- *for each state* s *in* $\mathcal{C}$ *that is not a canonical form with respect to* Δ *modulo* B, $s \to^{+}_{\Delta,B} s{\downarrow}_{(\Delta,B)} \models \mathcal{F}(\mathcal{A})$.

To check an assertional specification $\mathcal{A}$ in a given computation $\mathcal{C}$, we can simply traverse $\mathcal{C}$ and progressively evaluate system assertions over simplified states and functional assertions over state equational simplifications, respectively. Definition 8 formalizes this methodology into the function $check(\mathcal{C},\mathcal{A})$ that takes as input a computation $\mathcal{C}$ and an assertional specification $\mathcal{A}$ and delivers a triple $(\mathcal{P}, Err, flag)$ where $\mathcal{P}$ is a prefix of $\mathcal{C}$, *Err* is a set of functional or system error symptoms with respect to $\mathcal{A}$, and $flag \in \{none, sys, fun\}$.

Roughly speaking, function $check(\mathcal{C},\mathcal{A})$ returns $(\mathcal{P}, Err, flag)$ as soon as it encounters either a state or a state equational simplification in which $\mathcal{A}$ is not satisfied: $\mathcal{P}$ represents a prefix of $\mathcal{C}$ that reaches a state in which a system/functional assertion is violated, *Err* specifies the associated error symptom set, and *flag* declares the nature of the computed symptoms (*fun* stands for functional error symptoms, *sys* for system error symptoms, and the keyword *none* indicates that no symptom has been identified).

Definition 8 (assertion checking)**.** *Let* $\mathcal{R} = (\Sigma, E, R)$, *with* $E = \Delta \cup B$, *be a rewrite theory and* $\mathcal{C}$ *be a computation in* $\mathcal{R}$. *Let* $\mathcal{A}$ *be an assertional specification for* $\mathcal{R}$.

$$check(\mathcal{C},\mathcal{A}) = \begin{cases} (s{\downarrow}_{(\Delta,B)}, \emptyset, none) & \textit{if } \mathcal{C} = s{\downarrow}_{(\Delta,B)} \textit{ and } s{\downarrow}_{(\Delta,B)} \models \mathcal{S}(\mathcal{A}) \\ (s{\downarrow}_{(\Delta,B)}, \xi_{sys}(s, \mathcal{S}(\mathcal{A})), sys) & \textit{if } \mathcal{C} = s{\downarrow}_{(\Delta,B)} \textit{ and } s{\downarrow}_{(\Delta,B)} \not\models \mathcal{S}(\mathcal{A}) \\ (\mu \to^{*}_{R,B} \mathcal{C}'', Err, flag) & \textit{if } \mathcal{C} = \mu \to^{*}_{R,B} \mathcal{C}' \textit{ and } \mu \models \mathcal{F}(\mathcal{A}) \\ & \textit{and } (\mathcal{C}'', Err, flag) = \\ & check(Can(\mu) \to^{*}_{R,B} \mathcal{C}', \mathcal{A}) \\ (\mu, \xi_{fun}(\mu, \mathcal{F}(\mathcal{A})), fun) & \textit{if } \mathcal{C} = \mu \to^{*}_{R,B} \mathcal{C}' \textit{ and } \mu \not\models \mathcal{F}(\mathcal{A}) \\ (s \to_{R,B} \mathcal{C}'', Err, flag) & \textit{if } \mathcal{C} = s \to_{R,B} \mathcal{C}',\ s = s{\downarrow}_{(\Delta,B)} \textit{ and} \\ & s \models \mathcal{S}(\mathcal{A}) \textit{ and } (\mathcal{C}'', Err, flag) = \\ & check(\mathcal{C}', \mathcal{A}) \\ (s, \xi_{sys}(s, \mathcal{S}(\mathcal{A})), sys) & \textit{if } \mathcal{C} = s \to_{R,B} \mathcal{C}',\ s = s{\downarrow}_{(\Delta,B)} \\ & \textit{and } s \not\models \mathcal{S}(\mathcal{A}) \end{cases}$$

where $\mu = s \to^{+}_{\Delta,B} s{\downarrow}_{\Delta,B}$ *is a non-empty equational simplification for* s *and* $Can(\mu) = s{\downarrow}_{\Delta,B}$.

Example 14. *Consider the rewrite theory* $\mathcal{R}$ *of Example 5 together with the assertional specification* $\mathcal{A}$ *and the state equational simplification* $\mu_{rent} = s \to^{+}_{\Delta,B} s{\downarrow}_{\Delta,B}$

of Example 13. Recall that μ_{rent} erroneously suspends the preferred customer `'C2` *through the equational simplification $\mu_{'\mathtt{C2}}$ included in μ_{rent}. This error is pinpointed by the refutation of the functional assertion $\Phi \in \mathcal{A}$.*

Now, by Definition 8,

$$check(\mu_{rent}, \mathcal{A}) = (\mu_{rent}, \ \{(\mu_{'\mathtt{C2}}, \{3.2.1\})\}, fun),$$

since $\mu_{rent} \not\models \mathcal{F}(\mathcal{A})$ where

$$\mathcal{F}(\mathcal{A}) = \{\Phi\} \text{ and } \xi_{fun}(\mu_{\mathtt{rent}}, \mathcal{F}(\mathcal{A})) = \{(\mu_{'\mathtt{C2}}, \{3.2.1\})\}.$$

The following proposition states that function *check* can be effectively used to dynamically check assertional specifications across computations.

Proposition 1. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and $\mathcal{C}$ be a computation in $\mathcal{R}$. Let $\mathcal{A}$ be an assertional specification for $\mathcal{R}$. Then, $\mathcal{C} \models \mathcal{A}$ iff $check(\mathcal{C}, \mathcal{A}) = (\mathcal{C}, \emptyset, none)$.*

Proof. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and $\mathcal{C}$ be a computation in $\mathcal{R}$. Let $\mathcal{A}$ be an assertional specification for $\mathcal{R}$.*

($\Longrightarrow$) *We assume that $\mathcal{A}$ is satisfied in $\mathcal{C}$, that is, $\mathcal{C} \models \mathcal{A}$. Hence, by Definition 7, for each state s in $\mathcal{C}$ that is a canonical form with respect to Δ modulo B, $s \models \mathcal{S}(\mathcal{A})$; and for each state s in $\mathcal{C}$ that is not a canonical form with respect to Δ modulo B, $s \rightarrow^{+}_{\Delta,B} s{\downarrow}_{(\Delta,B)} \models \mathcal{F}(\mathcal{A})$. We now proceed by induction on the length of the computation $\mathcal{C}$.*

Base case: *$\mathcal{C} = s_0{\downarrow}_{\Delta,B}$. In this case $\mathcal{C}$ consists of the single initial state $s_0{\downarrow}_{\Delta,B}$, which is a canonical form with respect to Δ modulo B. Since $\mathcal{A}$ is satisfied in $\mathcal{C}$, we have $s_0{\downarrow}_{\Delta,B} \models \mathcal{S}(\mathcal{A})$. Hence, by Definition 8, we trivially have*

$$check(\mathcal{C}, \mathcal{A}) = check(s_0{\downarrow}_{\Delta,B}, \mathcal{A}) = (s_0{\downarrow}_{\Delta,B}, \emptyset, none) = (\mathcal{C}, \emptyset, none).$$

Inductive case 1: *$\mathcal{C} = (s_0 \rightarrow^{+}_{\Delta,B} s_0{\downarrow}_{\Delta,B} \rightarrow^{*}_{R,B} \mathcal{C}')$. In this case, $\mathcal{C}$ is a (non-empty) computation that initially simplifies the non-normalized input term s_0 by means of the equational state simplification $\mu = (s_0 \rightarrow^{+}_{\Delta,B} s_0{\downarrow}_{\Delta,B})$. By inductive hypothesis, we have that*

$$check(s_0{\downarrow}_{\Delta,B} \rightarrow^{*}_{R,B} \mathcal{C}', \mathcal{A}) = (\varepsilon, \emptyset, none).$$

Furthermore, $\mu \models \mathcal{F}(\mathcal{A})$ since $\mathcal{A}$ is satisfied in $\mathcal{C}$ (and hence in μ). Therefore, $check(\mathcal{C}, \mathcal{A}) = check(\mu \rightarrow^{*}_{R,B} \mathcal{C}', \mathcal{A}) = (\mu \rightarrow^{*}_{R,B} \mathcal{C}', \emptyset, none) = (\mathcal{C}, \emptyset, none)$.

Inductive case 2: $\mathcal{C} = s \rightarrow_{R,B} \mathcal{C}'$. *Since the first rewrite step in* $\mathcal{C}$ *is a rule application, this implies that* s *is already in canonical form, that is,* $s = s{\downarrow_{\Delta,B}}$. *Since* $\mathcal{A}$ *is satisfied in* $\mathcal{C}$ *we have that* $s \models \mathcal{S}(\mathcal{A})$, *which implies*

$$s{\downarrow_{\Delta,B}} \models \mathcal{S}(\mathcal{A}). \tag{5.1}$$

Also, by inductive hypothesis, it holds that

$$\mathit{check}(\mathcal{C}', \mathcal{A}) = (\varepsilon, \emptyset, \mathit{none}). \tag{5.2}$$

By combining Claims 5.1 and 5.2, we get

$$\mathit{check}(\mathcal{C}, \mathcal{A}) = \mathit{check}(s \rightarrow_{R,B} \mathcal{C}', \mathcal{A}) = (s \rightarrow_{R,B} \mathcal{C}', \emptyset, \mathit{none}) = (\mathcal{C}, \emptyset, \mathit{none}).$$

($\Longleftarrow$) *By contradiction, we assume that* $\mathit{check}(\mathcal{C}, \mathcal{A}) = (\mathcal{C}, \emptyset, \mathit{none})$ *and* $\mathcal{C} \not\models \mathcal{A}$. *Since* $\mathcal{A}$ *is not satisfied in* $\mathcal{C}$, *there exists either a state* s *in canonical form such that* $s \not\models \mathcal{S}(\mathcal{A})$ *or an equational state simplification* μ *such that* $\mu \not\models \mathcal{F}(\mathcal{A})$. *Thus, by Definition 8, the function call* $\mathit{check}(\mathcal{C}, \mathcal{A})$ *delivers a triple* $(\mathcal{C}'', \mathit{Err}, \mathit{flag})$ *with* $\mathit{Err} \neq \emptyset$. *This leads to a contradiction since we have assumed* $\mathit{check}(\mathcal{C}, \mathcal{A}) = (\mathcal{C}, \emptyset, \mathit{none})$.

The runtime checking methodology formalized in Definition 8 can be interpreted either as an asynchronous (and trace-storing) technique or as a synchronous one (by considering that the input trace $\mathcal{C}$ is lazily generated as successive Maude steps that are incrementally consumed by the calculus). In the following chapter, we formalize a truly synchronous methodology where computations, or rather whole search trees, can be stepwisely examined in a forward direction, reporting a violation at the exact step where it occurs.

Chapter 6

Automated Debugging of Maude Programs

Dynamic assertion-checking and trace slicing can be smoothly combined together to facilitate the debugging of ill-defined rewrite theories. In the case when a functional or system assertion $A \in \mathcal{A}$ fails to be satisfied over a computation $\mathcal{C}$, a fragment of $\mathcal{C}$ (which exhibits the anomalous behavior with respect to A) is returned together with the corresponding set of system/functional error symptoms. Then, a backward trace slicing technique can take advantage of the computed error symptoms to produce small, easy-to-inspect computation slices of all those fragments that have been proven to be erroneous by the assertion-checking methodology. In conventional program development, if an assertion evaluates to false at runtime, an assertion failure is signaled, which typically causes execution to abort while delivering a huge execution trace. By automatically inferring deft slicing criteria from falsified assertions, the proposed methodology derives a self-initiating, enhanced dynamic slicing technique that automatically starts slicing the trace backwards at the time the assertion violation occurs, without having to manually determine the slicing criterion in advance.

The chapter is organized as follows. Section 6.1 introduces a backward trace slicing technique that can be used to drastically reduce complex, textually-large system computations with respect to user-defined slicing criteria that selects those data that we want to track back from a given point. Section 6.2 presents a technique that is able to improve the accuracy of the inferred error symptoms as well as the efficiency of the runtime checks by refining the formulas in the assertions. Section 6.3 formalizes the technique that automatizes debugging by combining runtime verification and backward trace slicing. Finally, Section 6.4 complements the methodology by introducing a new technique that automatically suggests suitable repairs to the rules involved in the violation of a system assertion.

6.1 Slicing of Execution Traces and Programs

Trace slicing [Alpuente et al., 2011, Alpuente et al., 2012, Alpuente et al., 2013a, Alpuente et al., 2014a] is a transformation technique for rewriting logic theories that can drastically reduce the size and complexity of entangled, textually-large execution traces by focusing on selected computation aspects. This is done by uncovering data dependences among related parts of the trace with respect to a user-defined slicing criterion (i.e., a set of symbols that the user wants to observe). This technique aims to improve the analysis, comprehension, and debugging of sophisticated rewrite theories by helping the user inspect involved traces in an easier way. By step-wisely reducing the amount of information in the simplified trace, it is easier for the user to locate program faults because pointless information or unwanted rewrite steps have been automatically removed. Roughly speaking, irrelevant terms in trace slices are omitted, leaving "holes" that are denoted by special variable symbols $\bullet$.

A term *slice* of the term s is a term $s^\bullet$ that hides part of the information in s; that is, the irrelevant data in s that we are not interested in are simply replaced by (fresh) $\bullet$-variables of appropriate sort, denoted by $\bullet_i$, with $i = 0, 1, 2, \ldots$.

The next auxiliary definition formalizes the function $\mathit{Tslice}(t, P)$, which allows a term slice of t to be constructed with respect to a set of positions P of t. The function *Tslice* relies on the function $\mathit{fresh}^\bullet$ whose invocation returns a (fresh) variable $\bullet_i$ of appropriate sort that is distinct from any previously generated variable $\bullet_j$.

Definition 9 (term slice)**.** *Let* $t \in \mathcal{T}(\Sigma, \mathcal{V})$ *be a term and let* P *be a set of positions such that* $P \subseteq \mathcal{P}os(t)$. *Then, the term slice* $\mathit{Tslice}(t, P)$ *of* t *with respect to* P *is computed as follows.*

$$\mathit{Tslice}(t, P) = \mathit{recslice}(t, P, \Lambda), \ \textit{where}$$

$$\mathit{recslice}(t, P, p) = \begin{cases} f(\mathit{recslice}(t_1, P, p.1), \ldots, \mathit{recslice}(t_n, P, p.n)) & \\ & \textit{if } t = f(t_1, \ldots, t_n), n \geq 0, \textit{and } p \in \bar{P} \\ t & \textit{if } t \in \mathcal{V}ar \textit{ and } p \in \bar{P} \\ \mathit{fresh}^\bullet & \textit{otherwise} \end{cases}$$

and $\bar{P} = \{u \mid u \leq p \wedge p \in P\}$ *is the* prefix closure *of* P. *Note that the inductive case (*$n = 0$*) includes the case when f is a* 0*-ary function symbol; hence,* $f(\mathit{recslice}(\emptyset, P, p)) = f$.

Roughly speaking, the function $\mathit{Tslice}(t, P)$ yields a term slice of t with respect to a set of positions P that includes all (and only the) symbols of t occurring within the access paths from the root of t to each position in P, while the remaining information of t is abstracted by means of $\bullet$-variables.

Example 15. *Consider the specification of Section 4.1.2 and the state*

```
t = < 'A1 : EconomyCar | available : true , rate : 20 >
      < 'RG : Register | rentals : 0 , date : 0 >
```

Consider the set $P = \{ 1.1,\ 1.2,\ 1.3.1,\ 1.3.2 \}$ *of positions in* t. *Then,*

$Tslice(t, P) =$ `< 'A1 : EconomyCar | available :` $\bullet_1$ `, rate :` $\bullet_2$ `>` $\bullet_3$

Trace slicing can be carried out forwards or backwards. While the forward trace slicing results in a form of impact analysis that identifies the scope and potential consequences of changing the program input, backward trace slicing allows provenance analysis to be performed; i.e., it shows how (parts of) a program output depend(s) on (parts of) its input and helps estimate which input data need to be modified to accomplish a change in the outcome. While dependency provenance provides information about the origins of (or influences upon) a given result, the notion of descendants is the key for impact evaluation. In the sequel, we focus on backward trace slicing.

Throughout this chapter, it is assumed the existence of a trace slicing function *backwardSlicing*$(s_0 \to^*_{\Delta\cup B} s_n, s^\bullet_n)$ as defined in [Alpuente et al., 2014a] that yields the backward trace slice $s^\bullet_0 \bullet\!\!\to^* s^\bullet_n$ of the computation trace $s_0 \to^*_{\Delta\cup B} s_n$ with respect to a term slice $s^\bullet_n$ of s_n, which is called the *slicing criterion*. This function relies on an instrumentation technique for Maude steps that allows the relevant information of the step, such as the selected redex and the contractum produced by the step, to be traced explicitly despite the fact that terms are rewritten modulo a set B of equational axioms (which may cause the components of the terms to be implicitly reordered in the original trace). Also, the dynamic dependencies exposed by backward trace slicing are exploited in [Alpuente et al., 2014a] to provide a (preliminary) program slicing capability that can identify those parts of a rewrite theory that can potentially affect the values computed at some point of interest.

Let us illustrate by means of an example how backward trace slicing works in practice and allows one to deduce the conditions under which a program produces the observed data.

Example 16. *Consider the* `RENT-A-CAR-ONLINE` *object module of Section 4.1.2 and the computation trace* $\mathcal{C}_{rent} = s_0 \xrightarrow{\texttt{3-day-rental}} s_1 \xrightarrow{\texttt{3-day-rental}} s_2$ *that starts in the initial state*

```
s0 = < 'A1 : EconomyCar | available : true , rate : 30 >
     < 'A5 : FullSizeCar | available : true , rate : 70 >
     < 'C1 : Customer | credit : 50 , suspended : false >
     < 'C2 : PreferredCustomer | credit : 100 , suspended : false >
     < 'RG : Register | date : 0 , rentals : 0 >
```

and ends in the state

```
s2 = < 'A1 : EconomyCar | available : false , rate : 30 >
     < 'A5 : FullSizeCar | available : false , rate : 70 >
     < 'C1 : Customer | credit : - 160 , suspended : false >
     < 'C2 : PreferredCustomer | credit : 10 , suspended : false >
     < 'R0 : Rental | car : 'A1 , customer : 'C2 , deposit : 90 ,
       dueDate : 3 , pickUpDate : 0 , rate : 30 >
     < 'R1 : Rental | car : 'A5 , customer : 'C1 , deposit : 210 ,
       dueDate : 3 , pickUpDate : 0 , rate : 70 >
     < 'RG : Register | date : 0 , rentals : 2 >
```

Roughly speaking, $\mathcal{C}_{rent}$ *models the two following actions*[1]*:*

(i) *customer* `'C2` *subscribes a 3-day rental contract (rule* `3-day-rental`*) to rent an economy car whose rate is* 30 *and his/her credit is reduced by* 90*,*

(ii) *customer* `'C1` *subscribes a 3-day rental contract (rule* `3-day-rental`*) to rent a full size car whose rate is* 70 *and his/her credit is reduced by* 210*.*

Let us assume we manually define as the slicing criterion the negative credit `- 160` *for customer* `'C1`*, which indicates a possible malfunction of the* `RENT-A-CAR` *specification since the regular client credit must be non-negative according to the semantics intended by the programmer. Therefore, we execute trace slicing on the trace* $\mathcal{C}_{rent}$ *with respect to the slicing criterion*

$$s_2^\bullet = \bullet_1\ \bullet_2\ \texttt{<}\ \bullet_3\ \texttt{:}\ \bullet_4\ \texttt{| credit : - 160 ,}\ \bullet_5\ \texttt{>}\ \bullet_6\ \bullet_7\ \bullet_8\ \bullet_9$$

that allows for the observation of `'C1`*'s negative credit. By applying the backward trace slicing technique of [Alpuente et al., 2014a] to* $\mathcal{C}_{rent}$ *with respect to* $s_2^\bullet$*, we get the output trace slice* $\mathcal{C}_{rent}^\bullet$*:*

$$\begin{array}{ll}
 & \bullet_{13}\ \texttt{<}\ \bullet_{10}\ \texttt{:}\ \bullet_{11}\ \texttt{| available : true , rate : 70 >} \\
 & \texttt{<}\ \bullet_3\ \texttt{:}\ \bullet_4\ \texttt{| credit : 50 , suspended : false >}\ \bullet_{14}\ \bullet_{15} \\
\xrightarrow{\texttt{3-day-rental}}_{\bullet} & \\
 & \bullet_1\ \texttt{<}\ \bullet_{10}\ \texttt{:}\ \bullet_{11}\ \texttt{| available : true , rate : 70 >} \\
 & \texttt{<}\ \bullet_3\ \texttt{:}\ \bullet_4\ \texttt{| credit : 50 , suspended : false >}\ \bullet_6\ \bullet_7\ \bullet_{12} \\
\xrightarrow{\texttt{3-day-rental}}_{\bullet} & \\
 & \bullet_1\ \bullet_2\ \texttt{<}\ \bullet_3\ \texttt{:}\ \bullet_4\ \texttt{| credit : - 160 ,}\ \bullet_5\ \texttt{>}\ \bullet_6\ \bullet_7\ \bullet_8\ \bullet_9
\end{array}$$

[1]For the sake of clarity, we have intentionally omitted, from $\mathcal{C}_{rent}$, all of the built-in equational simplifications that are needed to simplify arithmetic expressions.

Indeed, by observing the first sliced state in $\mathcal{C}^{\bullet}_{rent}$, we can easily verify that the conditions for the rental are met by customer `'C1` *and car* `'A5`. *In particular,* `'A5` *is available and (non-preferred) customer* `'C1` *is not suspended. However, the car should not be rented because the credit* `50` *does not cover the charge* `210` *(*`70` *for each day), which causes the negative credit* `- 160` *of customer* `'C1`.

The main idea of this work is to enhance backward trace slicing by using runtime assertion checking to automatically identify the relevant symbols to be traced back from the erroneous states of the trace, that is, those states where an assertion is falsified. In conventional program development environments, when a given assertion check fails, the programmer must thoughtfully identify which program statements impacted on the values that cause the assertion failure. An additional advantage of blending trace slicing and runtime checking together is that the runtime checking not only helps automate the trace slicing, but trace slicing also helps answer the “What caused it?” question that immediately arises when an assertion is violated. By using the assertion-based, backward trace slicing methodology, error diagnosis is greatly simplified because accurate criteria for slicing are automatically inferred from the computed error symptoms that initiate the slicing process so that much of the irrelevant data that does not influence the falsified assertions is automatically cut off.

6.2 Improving the Inference of the Slicing Criteria

This section presents a practical strategy that delivers finer slicing criteria by restricting the number of positions that are worth tracking to those that appear in selected subformulas of the (post)conditions.

Without loss of generality, it is assumed that any logic formula ϕ in a system assertion $S\,\{\varphi\}$ or in a functional assertion $I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ is written in conjunctive normal form $\varphi_1 \wedge \ldots \wedge \varphi_n$, where $\wedge$ does not occur in any φ_i, $i = 1, \ldots, n$. Then, a refined strategy for inferring accurate slicing criteria for an erroneous state e can be formulated as follows:

1. When a system assertion $S\ \{\varphi\}$ with $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_n$ is refuted, this is because e matches S (modulo the enriched equational theory) with matching substitution σ, while $\varphi\sigma$ is not satisfied. Hence, we sequentially examine the conjuncts φ_i, $i = 1, \ldots, n$, and for the first failing conjunct φ_j, a slicing criterion is synthesized by instantiating the pattern S with $\sigma_{\restriction \mathcal{V}ar(\varphi_j)}$, where $\sigma_{\restriction \mathcal{V}ar(\varphi_j)}$ is the restriction of σ to the variables that occur in φ_j.

2. In the case when a functional assertion $I\ \{\varphi_{in}\} \rightarrow O\ \{\varphi_{out}\}$ is violated, an instance $I\sigma_{in}$ of the input template I reduces to a canonical form O_e in e, and one of the following two cases occurs:

a) O_e does not equationally match the instance $O(\sigma_{in}\downarrow_{\Delta,B})\sigma_{out}$ of the output template O; in this case, the slicing criterion is computed by applying the modular order-sorted least general generalization algorithm of [Alpuente et al., 2014a] to gather up all of the mismatches (modulo the considered equational theory) between the erroneous canonical form O_e and $O(\sigma_{in}\downarrow_{\Delta,B})\sigma_{out}$.

b) O_e does equationally match $O(\sigma_{in}\downarrow_{\Delta,B})\sigma_{out}$, but all of the corresponding matchers falsify the formula φ_{out}; in this case, the methodology proceeds analogously to case 1 (system assertions) and synthesizes the slicing criterion by examining the failing conjuncts of φ_{out} systematically.

6.3 Integration of Assertion-Checking and Trace Slicing

Given a conditional rewrite theory $\mathcal{R} = (\Sigma, E, R)$, with $E = \Delta \cup B$, the transition space of all computations in $\mathcal{R}$ from the initial state s_0 can be represented as a *computation tree*,[2] $T_{\mathcal{R}}(s_0)$. Rewriting logic computation trees are typically large and complex objects that represent the highly-concurrent, non-deterministic nature of rewrite theories.

Let us formalize a methodology that checks rewrite theories with respect to an assertional specification $\mathcal{A}$ at runtime by incrementally generating and checking the computation tree $T_{\mathcal{R}}(s_0)$ until a fixed depth. In fact, the complete generation of $T_{\mathcal{R}}(s_0)$ is generally not feasible since some of its branches may be infinite as they encode non-terminating computations. The general analysis algorithm, which is specified by the routine $\mathit{analyze}(s_0, \mathcal{R}, \mathcal{A}, \mathit{depth})$, is given in Figure 6.1. The computation tree is constructed breadth-first, starting from a tree T that consists of a single root node s_0. At each expansion stage, the leaf nodes of the current T are computed by the function $\mathit{frontier}(T)$. Expansion of an arbitrary node s is done by deploying all the possible Maude steps stemming from s that are given by $m\mathcal{S}(s)$. Whenever a Maude step $\mathcal{M}$ is produced, it is also checked with respect to the specification $\mathcal{A}$ by calling $\mathit{check}(\mathcal{M}, \mathcal{A})$ that computes the triple $(\mathcal{P}, \mathit{Err}, \mathit{flag})$. According to the computed *flag* value, the algorithm distinguishes the following cases:

$\mathit{flag} = \mathit{none}$**.** No error symptoms have been computed; hence, $\mathcal{A}$ is satisfied in the Maude step $\mathcal{M}$, which can safely expand the node s by replacing s with the path represented by $\mathcal{M}$ via the invocation of $\mathit{add}(T, s, \mathcal{M})$, thereby augmenting T.

[2]In order to facilitate trace inspection, computations are visualized as trees, although they are internally represented by means of more efficient graph-like data structures that allow common subexpressions to be shared.

```
function $analyze(s_0, (\Sigma, \Delta \cup B, R), \mathcal{A}, depth)$
1.  $T = s_0$
2.  $d = 0$
3.  while $(d \leq depth)$ do
4.    $F = frontier(T)$
5.    for each $s \in F$
6.      for each $\mathcal{M} \in m\mathcal{S}(s)$
7.        $(\mathcal{P}, Err, flag) = check(\mathcal{M}, \mathcal{A})$
8.        case $flag$ of
9.          $none$ :
10.           $T = add(T, s, \mathcal{M})$
11.         $sys$ :
12.           $w = selectSysSymptom(Err)$
13.           $l^\bullet = TSlice(last(\mathcal{P}), \mathcal{P}os_w(last(\mathcal{P})))$
14.           return $backwardSlice(s_0 \to^*_{R \cup \Delta, B} \mathcal{P}, l^\bullet)$
15.         $fun$:
16.           $(t \to^+_{\Delta,B} t{\downarrow_{\Delta,B}}, L) = selectFunSymptom(Err)$
17.           $(t{\downarrow_{\Delta,B}})^\bullet = TSlice(t{\downarrow_{\Delta,B}}, \bigcup_{w \in L} \mathcal{P}os_w(t{\downarrow_{\Delta,B}}))$
18.           return $backwardSlice(t \to^+_{\Delta,B} t{\downarrow_{\Delta,B}}, (t{\downarrow_{\Delta,B}})^\bullet)$
19.       end case
20.     end for
21.   end for
22.   $d = d + 1$
23. end while
24. return $T$
end
```

Figure 6.1: The *analyze* function.

$flag = sys$**.** In this case, *check* returns a set of system error symptoms *Err* together with a computation $\mathcal{P}$ (which is a prefix of the Maude step $\mathcal{M}$) that violates a system assertion of $\mathcal{A}$. The computation $s_0 \to^*_{R \cup \Delta, B} \mathcal{P}$ is then generated and backward sliced with respect to a term slice $l^\bullet$ of the last state of $\mathcal{P}$. This term slice conveys all the relevant information that we automatically retrieve by using Definition 9 from the system error symptom w selected by the function $selectSysSymptom(Err)$, while all other symbols in l are considered meaningless and simply pruned away. This way, the algorithm delivers a trace slice $s_0^\bullet \bullet\!\!\to^* \mathcal{P}^\bullet$ that removes from the computation all the information that does not affect the production of the chosen error symptom.

$flag = fun$**.** Some functional assertions have been violated by the considered Maude step $\mathcal{M}$. Hence, the algorithm selects a functional error symptom $(t \rightarrow^{+}_{\Delta,B} t{\downarrow_{\Delta,B}}, L)$ and returns the backward trace slicing of $t \rightarrow^{+}_{\Delta,B} t{\downarrow_{\Delta,B}}$ with respect to a term slice of $t{\downarrow_{\Delta,B}}$ that includes all the subterms of $t{\downarrow_{\Delta,B}}$ that are rooted at positions in L. As explained in Section 5.3.2, these subterms indicate possible causes of the assertion violation.

It is worth noting that, in the proposed framework, no specific semantics is attached to *selectSysSymptom* and *selectFunSymptom* functions since many selection strategies can be specified with different degrees of automation and associated trade-offs. For instance, we can simply obtain a fully automatic selection strategy (which is the strategy followed by the ABETS tool of Chapter 7) by selecting the first symptom in *Err*. On the other hand, an interactive strategy can also be implemented by asking the user to choose a symptom at runtime.

Finally, if the *analyze* function terminates without detecting any assertion violation, then a (verified) tree T is delivered that encodes the first *depth* levels of the computation tree $T_{\mathcal{R}}(s_0)$; otherwise, the trace slice of the first computation that is found to violate an assertion is delivered. When multiple assertions are violated, *analyze* can be invoked iteratively: i.e., we can (manually) run *analyze* on a sequence of mutations of the original program that fix (if possible) the violations progressively encountered.

Theorem 1 (correctness). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $T_{\mathcal{R}}(s_0)$ be the computation tree for initial state $s_0 \in \mathcal{T}(\Sigma, \mathcal{V}ar)$ in $\mathcal{R}$, and $\mathcal{A}$ be an assertional specification for $\mathcal{R}$. Let depth be a natural number. Then, analyze$(s_0, \mathcal{R}, \mathcal{A}, depth)$ terminates and*

1. *if there exists a computation $\mathcal{C}$ in $T_{\mathcal{R}}(s_0)$ such that $\mathcal{C} \not\models \mathcal{A}$ and length$(\mathcal{C}) \leq$ depth, then analyze$(s_0, \mathcal{R}, \mathcal{A}, depth)$ delivers a backward trace slice $\mathcal{C}^{\bullet}_{pre}$ of a fragment $\mathcal{C}_{pre}$ of $\mathcal{C}$ that violates either a functional or a system assertion in $\mathcal{A}$. $\mathcal{C}^{\bullet}_{pre}$ is computed with respect to the term slice of the last state of $\mathcal{C}_{pre}$ that includes all subterms correlated to a chosen error symptom;*

2. *otherwise, analyze$(s_0, \mathcal{R}, \mathcal{A}, depth)$ delivers a tree T that corresponds to the expansion of the first depth levels of $T_{\mathcal{R}}(s_0)$.*

Proof (termination). *Termination of analyze$(s_0, \mathcal{R}, \mathcal{A}, depth)$ is trivial since the main* **while** *loop is performed at most depth times, and at each iteration it invokes the terminating backward slicing algorithm of [Alpuente et al., 2014a], and the function check, which is terminating because the execution in $\mathcal{R}$ of the assertional specification is also terminating.*

Now, let us prove Claim 1. Function analyze implements a breadth-first visit of the Maude steps in $T_{\mathcal{R}}(s_0)$ until an assertion violation occurs or the depth bound has been reached. Since we assume that there exists $\mathcal{C}$ in $T_{\mathcal{R}}(s_0)$ such that $\mathcal{C} \not\models \mathcal{A}$

and $length(\mathcal{C}) \leq depth$*, there exists a (minimum) prefix* $\mathcal{C}_{pre}$ *of* $\mathcal{C}$ *such that either* $\mathcal{C}_{pre} \not\models \mathcal{S}(\mathcal{A})$ *or* $\mathcal{C}_{pre} \not\models \mathcal{F}(\mathcal{A})$ *that is detected by analyze. Let us assume that* $\mathcal{C}_{pre} \not\models \mathcal{S}(\mathcal{A})$ *(the proof of the case* $\mathcal{C}_{pre} \not\models \mathcal{F}(\mathcal{A})$ *follows an analogous argument). Hence,* $\mathcal{C}_{pre} = s_0 \to^*_{\Delta\cup R,B} \mathcal{P}$ *where* $\mathcal{P}$ *is obtained by checking the last expanded Maude step* $\mathcal{M}$*, that is,* $check(\mathcal{M}, \mathcal{A}) = (\mathcal{P}, Err, sys)$*. Let* l *be the last state in* $\mathcal{C}_{pre}$*. Now, the state* l *is a canonical form such that* $l \not\models \Theta$ *for some* $\Theta \in \mathcal{S}(\mathcal{A})$*. Therefore,* $\xi_{sys}(l, \Theta) \subseteq Err$*. Let* $w \in \xi_{sys}(l, \Theta) \subseteq Err$ *be a selected system error symptom. By Definition 9,* $l^\bullet = TSlice(l, \mathcal{P}os_w(l))$ *computes a term slice* $l^\bullet$ *that includes all of the symbols in the subterm* $l|_w$*. Thus,*

$$backwardSlice(\mathcal{C}_{pre}, l^\bullet) = backwardSlice(s_0 \to^*_{R\cup\Delta,B} \mathcal{P}, l^\bullet)$$

is a backward trace slice of $\mathcal{C}_{pre}$ *that is computed with respect to a state* l *that includes the subterm* $l|_w$ *that is univocally correlated to the chosen system error symptom* w*.*

Note that, in the case when there is no computation $\mathcal{C}$ *in* $T_{\mathcal{R}}(s_0)$ *such that* $\mathcal{C} \not\models \mathcal{A}$ *and* $length(\mathcal{C}) \leq depth$*, Claim 2 is trivially proved by construction of the analyze function. In this case, there is no assertion violation, and thus the algorithm generates a tree T by unraveling all of the Maude steps of* $T_{\mathcal{R}}(s_0)$ *until the bound depth is reached.*

The assertion-based trace slicing methodology described in this section is a synchronous procedure that incrementally executes, checks, and possibly slices Maude computations at runtime. However, note that an offline, asynchronous procedure (that works on pre-calculated computations) can be easily derived from our synchronous algorithm with little effort. Actually, it suffices to provide the whole computation $\mathcal{C}$ to be analyzed as input and to stepwisely check its Maude steps by using the *check* function in search of assertion violations. When an assertion violation is detected on a prefix $\mathcal{C}_{pre}$ of the input computation that reaches the erroneous state e, a slicing criterion is then inferred by exploiting the error symptoms that are associated with the violation, as happened in the synchronous case; finally, a backward trace slice of $\mathcal{C}_{pre}$ is computed with respect to the considered slicing criterion. Synchronous and asynchronous modalities have been implemented in the ABETS tool, which is described in Chapter 7.

6.4 Automated Repair of Faulty Rules

In the proposed framework, when a system assertion is falsified the execution stops and the computation trace that led to the erroneous state is automatically simplified by means of a backward trace slicing technique. However, the user still has to *decipher* the counterexample and manually repair the program. In this regard, this section proposes a transformation technique that automatically computes repairs to the program by using equational unification and by constraining the rules involved in the generation of those erroneous states detected.

Given an equational theory $E = \Delta \cup Ax$ and two terms t_1 and t_2, an E-unifier for t_1 and t_2 is a substitution σ such that $t_1\sigma =_E t_2\sigma$. In Maude, E-unifiers are not represented as a single substitution, but as a pair of substitutions (σ_1, σ_2), one for left unificands and the other for right unificands (i.e., $t_1\sigma_1 =_E t_2\sigma_2$). Also, Maude's E-unification algorithm may generate new (fresh) unification variables, denoted by `%n`, with `n` being a natural number. The set of all such variables contained in a given term `t` is denoted by *UnifVar*$(\texttt{t})$. Let us see an example.

Example 17. *Consider a simple Maude program whose signature consists of two unary operators,* `m` *and* `c`*, and one commutative, binary operator* `f`*. The program includes a single equation* `m(X) = c(X)`.

Then,

$$\sigma = (\sigma_1, \sigma_2) = \texttt{(\{ X / \%1 \},\{ Z / \%1 \})}$$

is an E*-unifier for the terms* $t_1 = \texttt{f(m(X),0)}$ *and* $t_2 = \texttt{f(0,c(Z))}$*. The new, unification variable* `%1` *is used to establish that* `X` *and* `Z` *represent the same value, and it is the only common variable shared by* $t_1\sigma_1$ *and* $t_2\sigma_2$*.*

The proposed repair technique is based on a two-phase algorithm that takes as input:

(i) the last Maude step $s \xrightarrow{r,\sigma,w} t {\to}^*_{\Delta,B} t{\downarrow_{\Delta,B}}$ of the execution trace that violates $S\,\{\varphi\}$,

(ii) the violated system assertion $S\,\{\varphi\}$, and

(iii) the bug position p in the last trace state $t{\downarrow_{\Delta,B}}$.

Phase 1 [Semantic unification of the failing assertion and rule]. First we E-unify the terms $t[\rho]_w$ (that is, a more general version of $t = s[\rho\sigma]_w$ that does not apply the substitution σ to the reduced term) and $t{\downarrow_{\Delta,B}}\,[S]_p$ (that is, a more general version of $t{\downarrow_{\Delta,B}}$ where the subterm at the bug position p is replaced by the assertion pattern S itself) in order to relate the variables in the right-hand side ρ of r with the variables that appear in the state template S. Since there may be several E-unifiers, we just select an E-unifier (σ_ρ, σ_S) such that the bindings in σ_ρ do not clash with the bindings in the computed substitution σ. This is done by performing a standard consistency check through the parallel composition of σ_ρ and σ, which computes the most general unifier (mgu) of the set of all the equations $x = \mathit{term}$ that represent a binding $x\,/\,\mathit{term}$ in either σ_ρ or σ. If such an mgu exists, σ_ρ is consistent w.r.t. σ, and the corresponding E-unifier (σ_ρ, σ_S) is selected.

As an important remark, observe that we cannot simply E-unify ρ with S because the state template S could include operators that are not in ρ but in t, and, hence, the two terms could be not E-unifiable and lead to no repair. This is the reason why we need to E-unify ρ and S within their corresponding state contexts, that is, $t[\rho]_w$ and $t{\downarrow_{\Delta,B}}\,[\mathsf{S}]_p$.

Example 18. *Consider a Maude program that contains the rewrite rule*

```
rl [r] : f(X) => g(X) .
```

and no equations, together with the execution trace

$$\texttt{a \& f(0)} \xrightarrow{\texttt{r}} \texttt{a \& g(0)}$$

and the system assertion

```
(a & g(Z)) { Z > 0 }
```

that is violated in the state `a & g(0)`*. Observe that there is no* E*-unifier between the right-hand side* `g(X)` *of* `r` *and the state template* `a & g(Z)`*, whereas the pair* `({ X / %1 },{ Z / %1 })` *is an* E*-unifier for the terms* `a & g(X)` *and* `a & g(Z)`*, which include* `g(X)` *and* `a & g(Z)` *in their corresponding state context. More importantly, the bindings in the computed* E*-unifier enforce* `X` *and* `Z` *to bind the very same value. This suggests that we can achieve a repair by forcing the rewrite rule argument* `X` *to inherit the constraints on* `Z`*.*

Phase 2 [Strengthening the rule condition]. Given the computed E-unifier (σ_ρ, σ_S), first we split σ_ρ into two sets σ_{rule} and σ_{new} such that $\sigma_{rule} = \{\mathtt{X} \,/\, t \in \sigma_\rho \mid \mathtt{X} \in \mathcal{V}ar(\rho) \wedge \mathit{Unif}\mathcal{V}ar(t) = \emptyset\}$, and $\sigma_{new} = \sigma_\rho \setminus \sigma_{rule}$. Note that σ_{new} contains all those σ_ρ bindings that introduce new unification variables, while the bindings of σ_{rule} only use the original variables of ρ. Then, we replace the faulty rule `r` with the following corrected rule whose condition is strengthened by adding a constrained version (that is built by using σ_S and σ_{rule}) of the violated logic formula φ:

$$\texttt{crl [rfix] :}\ \lambda\sigma_{new}\ \texttt{=>}\ \rho\sigma_{new}$$
$$\texttt{if}\ C\sigma_{new}\ \texttt{/\textbackslash}\ \Big(\big(\bigwedge_{\mathtt{X}/t\in\sigma_{rule}} \mathtt{X} == t\big)\ \texttt{implies}\ \varphi\sigma_S\Big).$$

The corrected rule `rfix` is produced by instantiating the original rule `r` with the substitution σ_{new} that introduces in `rfix` the fresh variables generated during the unification process of Phase 1 and by adding the instance $\varphi\sigma_S$ of the falsified logical formula φ. The variables of such an instance are constrained via a logical implication whose premise is the conjunction of all the bindings $\mathtt{X}\ /\ t$ in σ_{rule} interpreted as

Boolean expressions X == t[3]. In the case when $\sigma_{\mathtt{rule}}$ is empty, the logical implication corresponds to (`true implies` $\varphi\sigma_S$), and thus simply reduces to the term $\varphi\sigma_S$.

Example 19. *Consider a Maude program that includes the following conditional rewrite rule* `r` *and equation* `e`

```
crl [r] : f(X,Y) => c(2,g(X,Y)) if X =/= Y .
eq  [e] : g(X,Y) = m(X,Y) .
```

and assume that the operator `m` *is declared commutative. Also consider the system assertion* `c(2,m(Z,5)) {even(Z)}`, *where* `even(Z)` *checks if* `Z` *is an even natural number.*

The execution trace $\mathtt{f}(5,3) \xrightarrow{r,\sigma} \mathtt{c}(2,\mathtt{g}(5,3)) \xrightarrow{e} \mathtt{c}(2,\mathtt{m}(5,3))$, *with computed substitution* $\sigma = \{\mathtt{X}\ /\ 5,\ \mathtt{Y}\ /\ 3\}$, *is erroneous since the formula* `even(Z)` *does not hold for the binding* `Z / 3` *that is computed by matching modulo commutativity the state* `c(2,m(5,3))` *in the assertion state template* `c(2,m(Z,5))`.

The repair proceeds by first performing Phase 1, which computes two E*-unifiers of the terms* `c(2,g(X,Y))` *and* `c(2,m(Z,5))`, *namely,*

$$(\sigma_{\rho_1}, \sigma_{S_1}) = (\{\mathtt{X}\ /\ \%1,\ \mathtt{Y}\ /\ 5\}, \{\mathtt{Z}\ /\ \%1\})$$
$$(\sigma_{\rho_2}, \sigma_{S_2}) = (\{\mathtt{X}\ /\ 5,\ \mathtt{Y}\ /\ \%1\}, \{\mathtt{Z}\ /\ \%1\})$$

Now, observe that the E*-unifier* $(\sigma_{\rho_1}, \sigma_{S_1})$ *is discarded since* σ_{ρ_1} *is not consistent w.r.t.* σ. *Actually, there is no mgu of* σ_{ρ_1} *and* σ *because of the clash between the bindings* $\mathtt{Y}\ /\ 5 \in \sigma_{\rho_1}$ *and* $\mathtt{Y}\ /\ 3 \in \sigma$. *The* E*-unifier* $(\sigma_{\rho_2}, \sigma_{S_2})$ *is consistent w.r.t.* σ *and thus is used to infer the repair in Phase 2 of the algorithm.*

Phase 2 generates the partition $\sigma_{\rho_2} = \sigma_{\mathtt{rule}} \cup \sigma_{new} = \{\mathtt{X}\ /\ 5\} \cup \{\mathtt{Y}\ /\ \%1\}$ *and uses it together with* σ_{S_2} *to yield the following corrected version of the rule* `r`*:*

```
crl [rfix] : f(X,%1) => c(2,g(X,%1))
             if (X =/= %1 / (X == 5 implies even(%1)) .
```

Note that the generated condition of a repaired rule `rfix` might not be satisfiable, which makes `rfix` not applicable. Nevertheless, this is not bad since the non-applicability of the corrected rule prevents the system from reaching the faulty state signaled by the assertion violation. This therefore has the inherent effect of reducing the number of erroneous runs in the system, which is of primary importance in the repair of critical systems as first advocated by [Logozzo and Ball, 2012].

[3] A binding X / t in $\sigma_{\mathtt{rule}}$ can always be interpreted as an executable, Boolean expression X == t, since all the variables included in X / t appear in the rewrite rule as well and thus take concrete values when the rule is applied.

Chapter 7

The ABETS System

The assertion-based, runtime checking, analysis, and repair methodologies presented in this thesis have been implemented in the prototype tool ABETS (*Assertion-BasEd Trace Slicer*), which is publicly available at the ABETS website at `http://safe-tools.dsic.upv.es/abets`. For implementing the exploration capabilities of ABETS, part of the inspection machinery of the dynamic exploration framework ANIMA, which was developed in previous work [Alpuente et al., 2015a], was reused. Likewise, the slicing-based analysis capabilities of ABETS (and ANIMA) are rooted in the trace (and program) slicing procedures developed in [Alpuente et al., 2014a], which were first implemented in the dynamic slicing tool *i*Julienne [Alpuente et al., 2013a]. One of the main novelties of ABETS with respect to previous (ANIMA and *i*Julienne) systems is that it has been implemented to run at both the Core Maude and Full Maude levels. Furthermore, the need to invoke Full Maude is automatically inferred so that high-performance analyses can be achieved for theories that do not require the time-expensive, Full Maude capabilities. This improvement was then ported back to ANIMA and *i*Julienne as well.

This chapter presents ABETS, an assertion-based, automated dynamic trace slicer that implements the methodologies presented in Chapters 5 and 6 of this thesis. The structure of this chapter is as follows. Section 7.1 presents the main features of ABETS. Section 7.2 reproduces a debugging session with ABETS. Section 7.3 describes the most relevant implementation details, including both the architecture of ABETS and some significant optimizations. Finally, Section 7.4 collects some experiments that benchmark the efficiency of the system.

7.1 ABETS in a Nutshell

This section briefly describes the ABETS system, which operates under three different modes that are described in Section 7.1.1, while Section 7.1.2 explains the most relevant services provided by the system.

7.1.1 Operating Modes

Given a Maude module that encodes the rewrite theory $\mathcal{R}$, runtime assertion checking is performed in ABETS by first wrapping $\mathcal{R}$, via inclusion, in a system module PRED($\mathcal{R}$) that also contains the extra predicates the user may need to define new formulas. Functional and system assertions are given sorts in the functional module ASSERTION

```
fmod ASSERTION is
  sorts sAssertion fAssertion Assertion .
  subsorts sAssertion fAssertion < Assertion .
  op _/\_ : Bool Bool -> Bool [ctor assoc prec 125 gather (e e)] .
  op _`{_`} : Universal Bool -> sAssertion [ctor poly (1)] .
  op _`{_`}->_`{_`} : Universal Bool Universal Bool -> fAssertion
     [ctor poly (1 3)] .
endfm
```

which is also included in PRED($\mathcal{R}$). This hierarchical setup allows assertions specifications in PRED($\mathcal{R}$) to be directly parsed by means of Maude's `metaParse` operation, resulting in a list $\mathcal{A}$ of (system and functional) assertions.

Given a computation $\mathcal{C}$ in $\mathcal{R}$, for asynchronous checking ABETS proceeds by incrementally consuming Maude steps (of $\mathcal{C}$) while checking the assertions in $\mathcal{A}$ that are relevant to the step.

In contrast, for the synchronous case, ABETS distinguishes two additional operation modes, depending on how computations are dynamically generated. The first mode consists of a step-by-step generation of a computation $\mathcal{C}$ that follows Maude's internal strategy. Each time a Maude step is generated, the satisfaction of $\mathcal{A}$ is checked, which is similar to the asynchronous case. The second mode allows a fragment of the whole computation tree to be deployed up to a given depth that is measured in Maude steps. Also in this case, the satisfaction of $\mathcal{A}$ is checked at each Maude step.

In the event that an assertion is falsified at state s_n, the dynamic checking is immediately stopped and ABETS delivers a (simplified) counter-example trace. As explained in Section 6.3, for system assertions violations the simplified trace is computed as the backward slicing of the trace from the initial state s_0 to s_n (with respect to a slicing criterion that is automatically inferred by matching the discordant subterm of s_n with the state pattern of the falsified assertion). As for functional assertions, the delivered counter-examples consist of the equational simplification trace for the outermost term that is responsible for the falsification. In this case, the slicing criteria is automatically obtained as the discrepancy between the normal form that is expected and the normal form that is actually computed. This discrepancy is calculated by using the least general generalization algorithm of [Alpuente et al., 2014d] that was first implemented in ACUOS [Alpuente et al., 2014e], which has been coupled into the ABETS core. Note that the asynchronous mode is preferable for the debugging of previously identified faulty executions, since this mode avoids having to re-execute the program and thus it is the lightest of all three checking modes.

While the synchronous tree-checking mode is useful for analyzing all the non-deterministic paths of a non-confluent program execution at the same time, the synchronous trace-checking mode is the best to debug deterministic executions or (arbitrarily chosen) non-deterministic computations of non-confluent programs. An upper bound is necessary to finitize the analyses in the synchronous checking modes, which is typically based on counting the elapsed time or the number of rewrite steps. For the synchronous trace-checking mode, ABETS has 250 rewrite steps as the upper bound, mainly because trace formatting is rather time-consuming (the instrumentation of a trace with 250 rewrite steps can result in thousands of instrumented rewrite steps that also need to be properly formatted to be output). Nevertheless, offline (console) checking can deal with much higher bounds, especially when formatting is dispensed. As for the synchronous tree-checking mode, ABETS includes two different bounds: the first one limits the depth of the deployed computation trees to 10, while the second one limits the number of nodes that can be checked to 100,000.

Finally, as it has been mentioned in Section 3.3.2, object-oriented modules are just syntactic sugar in Maude and are internally transformed into system modules for execution purposes. In object notation, object attributes do not need to be explicitly written in the rules when they remain unchanged, which overcomes the classical annoyance of expressing invariance or *frame properties* in algebraic specifications (i.e., that those parts of a state that are not affected by a change remain unchanged). However, these attributes do appear in (desugared) program states and computation traces. In order to simplify object trace slices to the fullest and to effectively deal with frame properties by mimicking attribute hiding, ABETS is endowed with refined matching and filtering procedures that are transparent to the user and that are automatically activated when dealing with object modules.

7.1.2 Features

This section summarizes the most relevant features and functionalities provided by ABETS.

Constraint Checking

ABETS implements the analysis technique formalized in Chapters 5 and 6 in both the asynchronous modality and the two synchronous modes previously described.

Automated Slicing

The tool is endowed with a (forward and backward) incremental, interactive trace slicer, which allows the user to greatly simplify any execution trace that falsifies at least one of the constraints. In order to incrementally unmask the bugs that are responsible for the errors, at each run, the slicing criterion is automatically inferred with respect to the first non-satisfied constraint.

Criteria Filtering

ABETS also provides a handy way to automatically synthesize refined slicing criteria by means of special variables (i.e., those whose names begin with #) that can be used in the assertions to indicate pieces of the matched term that the user does not want to observe along the generated trace slice.

Incremental Trace Slicing

ABETS is equipped with an incremental backward trace slicing algorithm that allows the computed trace slices to be further simplified by automatically applying backward as well as forward trace slicing with respect to user-provided slicing criterion refinements [Alpuente et al., 2015a].

Program Slicing

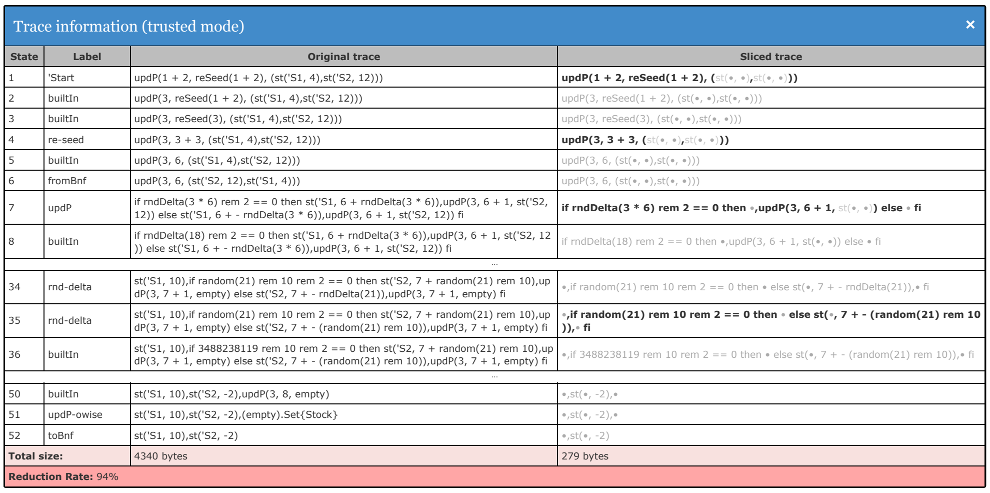
In addition to the simplification achieved by slicing execution traces, ABETS offers the user the possibility to compute a dynamic *program slice* that only contains the potentially faulty rules or equations [Alpuente et al., 2014a]. This feature is particularly useful in the case when a functional assertion fails, since the relevant equations are isolated from a presumably large, complex program that may possibly consist of many modules.

Autorepair of Rules

ABETS is also provided with an automated program repair facility, which is described in Section 6.4, that suggests fixes to potentially buggy rewrite rules whenever it detects a faulty system state of a trace $\mathcal{T}$ that does not satisfy a system assertion $S\,\{\varphi\}$. Roughly speaking, the technique transforms the rewrite rule that is responsible for the system assertion failure (i.e., the last applied rule in $\mathcal{T}$ that causes (a piece of) the transformed state to match the state pattern S). This fix is done by adding a constrained instance of the logic formula φ into the conditional part of the rule, which is computed by using Maude's built-in E-unification [Durán et al., 2016].

Interactive Navigation

Computations and computation slices can be easily and thoroughly inspected by navigating the traces and by accessing all of their available information, which includes the details of the instrumentation of each Maude step. Specifically, for each instrumented Maude step, ABETS shows the rule and equations possibly applied together with their computed matching substitutions, *redexes*, and *contractums*. All this information is accessible in both source and meta-level representations. Moreover, for conditional rewrite steps, an in-depth analysis of the condition proofs can be accessed through the *Inspect condition* option of the context menu.



Trace information (trusted mode)

State	Label	Original trace	Sliced trace
1	'Start	updP(1 + 2, reSeed(1 + 2), (st('S1, 4),st('S2, 12)))	updP(1 + 2, reSeed(1 + 2), (st(•, •),st(•, •)))
2	builtIn	updP(3, reSeed(1 + 2), (st('S1, 4),st('S2, 12)))	updP(3, reSeed(1 + 2), (st(•, •),st(•, •)))
3	builtIn	updP(3, reSeed(3), (st('S1, 4),st('S2, 12)))	updP(3, reSeed(3), (st(•, •),st(•, •)))
4	re-seed	updP(3, 3 + 3, (st('S1, 4),st('S2, 12)))	updP(3, 3 + 3, (st(•, •),st(•, •)))
5	builtIn	updP(3, 6, (st('S1, 4),st('S2, 12)))	updP(3, 6, (st(•, •),st(•, •)))
6	fromBnf	updP(3, 6, (st('S2, 12),st('S1, 4)))	updP(3, 6, (st(•, •),st(•, •)))
7	updP	if rndDelta(3 * 6) rem 2 == 0 then st('S1, 6 + rndDelta(3 * 6)),updP(3, 6 + 1, st('S2, 12)) else st('S1, 6 + - rndDelta(3 * 6)),updP(3, 6 + 1, st('S2, 12)) fi	if rndDelta(3 * 6) rem 2 == 0 then •,updP(3, 6 + 1, st(•, •)) else • fi
8	builtIn	if rndDelta(18) rem 2 == 0 then st('S1, 6 + rndDelta(3 * 6)),updP(3, 6 + 1, st('S2, 12)) else st('S1, 6 + - rndDelta(3 * 6)),updP(3, 6 + 1, st('S2, 12)) fi	if rndDelta(18) rem 2 == 0 then •,updP(3, 6 + 1, st(•, •)) else • fi
…			
34	rnd-delta	st('S1, 10),if random(21) rem 10 rem 2 == 0 then st('S2, 7 + random(21) rem 10),updP(3, 7 + 1, empty) else st('S2, 7 + - rndDelta(21)),updP(3, 7 + 1, empty) fi	•,if random(21) rem 10 rem 2 == 0 then • else st(•, 7 + - rndDelta(21)),• fi
35	rnd-delta	st('S1, 10),if random(21) rem 10 rem 2 == 0 then st('S2, 7 + random(21) rem 10),updP(3, 7 + 1, empty) else st('S2, 7 + - (random(21) rem 10)),updP(3, 7 + 1, empty) fi	•,if random(21) rem 10 rem 2 == 0 then • else st(•, 7 + - (random(21) rem 10)),• fi
36	builtIn	st('S1, 10),if 3488238119 rem 10 rem 2 == 0 then st('S2, 7 + random(21) rem 10),updP(3, 7 + 1, empty) else st('S2, 7 + - (random(21) rem 10)),updP(3, 7 + 1, empty) fi	•,if 3488238119 rem 10 rem 2 == 0 then • else st(•, 7 + - (random(21) rem 10)),• fi
…			
50	builtIn	st('S1, 10),st('S2, -2),updP(3, 8, empty)	•,st(•, -2),•
51	updP-owise	st('S1, 10),st('S2, -2),(empty).Set{Stock}	•,st(•, -2),•
52	toBnf	st('S1, 10),st('S2, -2)	•,st(•, -2)
Total size:		4340 bytes	279 bytes
Reduction Rate: 94%			

Figure 7.1: Computed trace slice after refuting the assertion of Example 9.

Trusted/Untrusted Modes

ABETS encompasses two slicing modes: trusted and untrusted. In trusted mode, Maude built-in operators are considered to be trusted (i.e., not to have bugs) and are therefore ignored in the trace slice (See Figure 7.1), which further reduces its size. In untrusted mode, all relevant operators are traced. The trusted mode is set to true by default and can be switched to untrusted mode by choosing the *Trace information* option in the main menu and then clicking the *Trusted/Untrusted mode* button. To help the user compare the original, extended trace and the trace slice when they are shown side-by-side (e.g., in the table view), trusted reduction steps (as well as duplicate states modulo axioms) are not omitted but depicted in light gray.

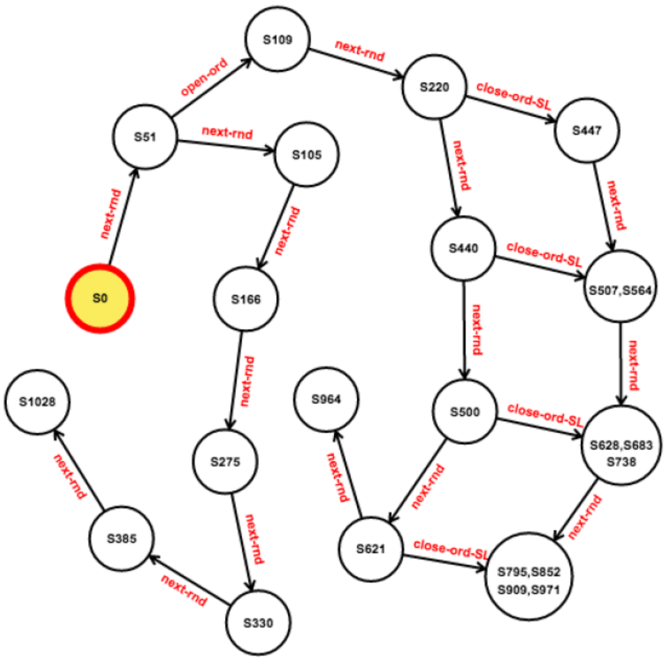


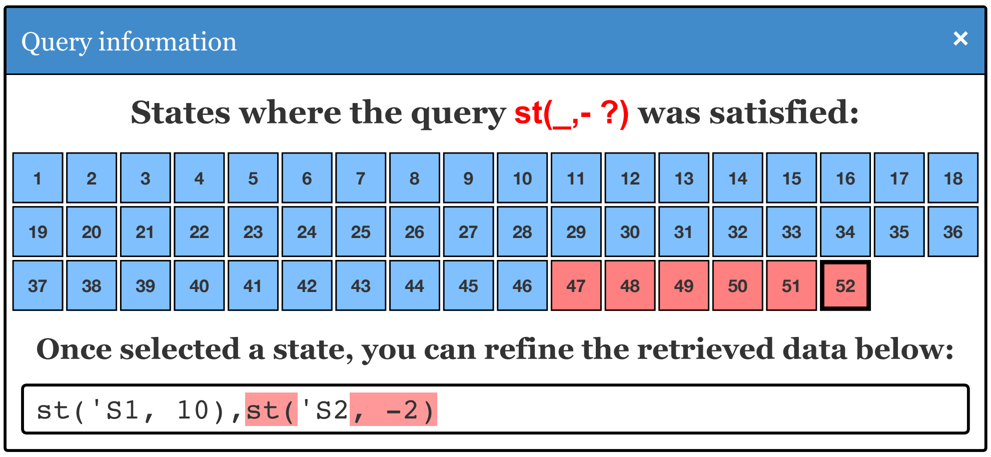
Figure 7.2: Computation graph generated from s_0 of Example 16.

Computation Graph Exploration

To help identify traces of interest for asynchronous checking, ABETS supports two different representations of the computation space for a given initial term: the (standard) tree representation that is provided by default and a novel graph representation of the state space that can improve user's understanding of the program behavior (see Figure 7.2). It is possible to switch between the two representations by left-clicking on any node of the tree or graph. In the case when the user left-clicks on a node in the graph, the topmost leftmost node in the tree that is associated with the considered graph node is highlighted.

Trace querying and manipulation

This feature allows information of interest to be searched in huge execution traces by undertaking a query that specifies a template for the search (see Figure 7.3). The query is a filtering pattern with wildcards that define irrelevant terms by means of the underscore character (_) and relevant terms by means of the question mark character (?). In addition, traces and trace slices can be manipulated using their meta-level rep-



```
st('S1, 10),st('S2, -2)
```

Figure 7.3: Result of the trace query `st(_, - ?)`.

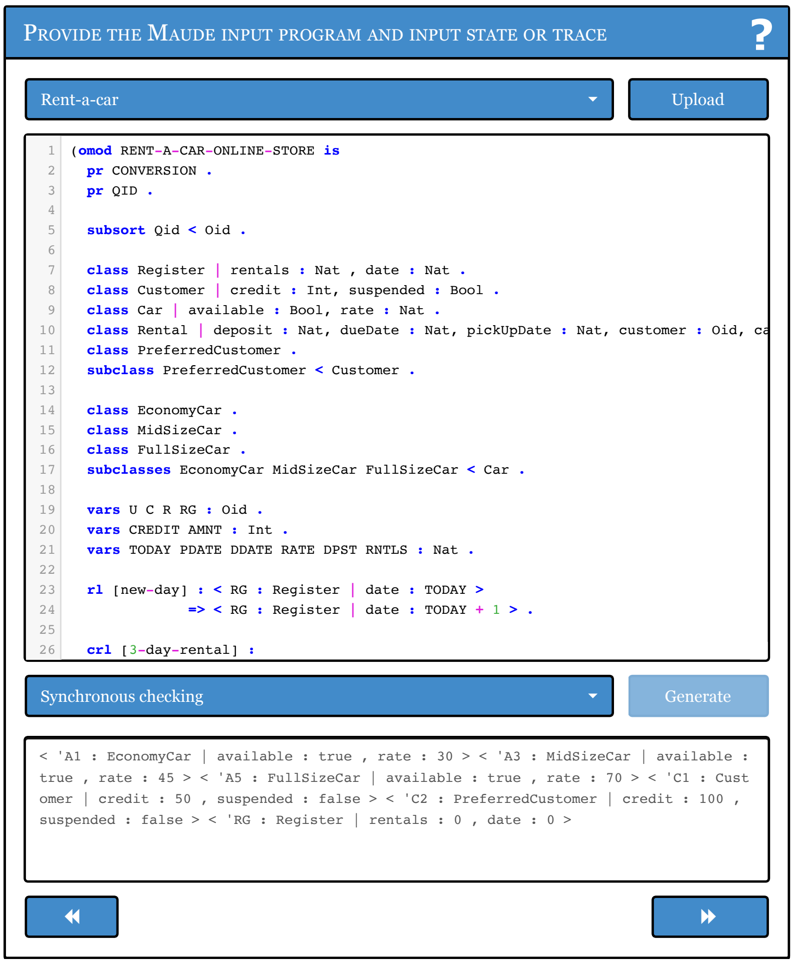
resentation to be exported to other Maude tools. The meta-representation of terms can be visually displayed, which is particularly useful for the analysis of object-oriented computations where some object attributes can only be unambiguously visualized in the meta-level (desugared) states.

7.2 ABETS at Work

Maude programs can be uploaded in ABETS as simple `.maude` or `.fm` files. Some predefined specifications are provided with the tool for demonstration purposes, including the car rental system of Section 4.1.2, and an assertional specification $\mathcal{A}_{\mathtt{rent}}$ that contains the system assertion Θ of Example 8 and the functional assertion Φ of Example 11. Let us consider the synchronous checking modality that (non-deterministically) expands all Maude steps that originate from the initial state:

```
s0 = < 'A1 : EconomyCar | available : true , rate : 30 >
     < 'A3 : MidSizeCar | available : true , rate : 45 >
     < 'A5 : FullSizeCar | available : true , rate : 70 >
     < 'C1 : Customer | credit : 50 , suspended : false >
     < 'C2 : PreferredCustomer | credit : 100 , suspended : false >
     < 'RG : Register | date : 0 , rentals : 0 >
```

This is achieved in ABETS by calling *analyze*$(\mathcal{R}_{\mathtt{rent}}, \mathcal{A}_{\mathtt{rent}}, 1)$, where $\mathcal{R}_{\mathtt{rent}}$ is the rewrite theory specified by the `RENT-A-CAR` object module. The screenshots shown in Figure 7.4 and Figure 7.5 illustrate the initial, input phase for the parameters $\mathcal{R}_{\mathtt{rent}}$ and $\mathcal{A}_{\mathtt{rent}}$, respectively.



```
(omod RENT-A-CAR-ONLINE-STORE is
  pr CONVERSION .
  pr QID .

  subsort Qid < Oid .

  class Register | rentals : Nat , date : Nat .
  class Customer | credit : Int, suspended : Bool .
  class Car | available : Bool, rate : Nat .
  class Rental | deposit : Nat, dueDate : Nat, pickUpDate : Nat, customer : Oid, ca
  class PreferredCustomer .
  subclass PreferredCustomer < Customer .

  class EconomyCar .
  class MidSizeCar .
  class FullSizeCar .
  subclasses EconomyCar MidSizeCar FullSizeCar < Car .

  vars U C R RG : Oid .
  vars CREDIT AMNT : Int .
  vars TODAY PDATE DDATE RATE DPST RNTLS : Nat .

  rl [new-day] : < RG : Register | date : TODAY >
              => < RG : Register | date : TODAY + 1 > .

  crl [3-day-rental] :
```

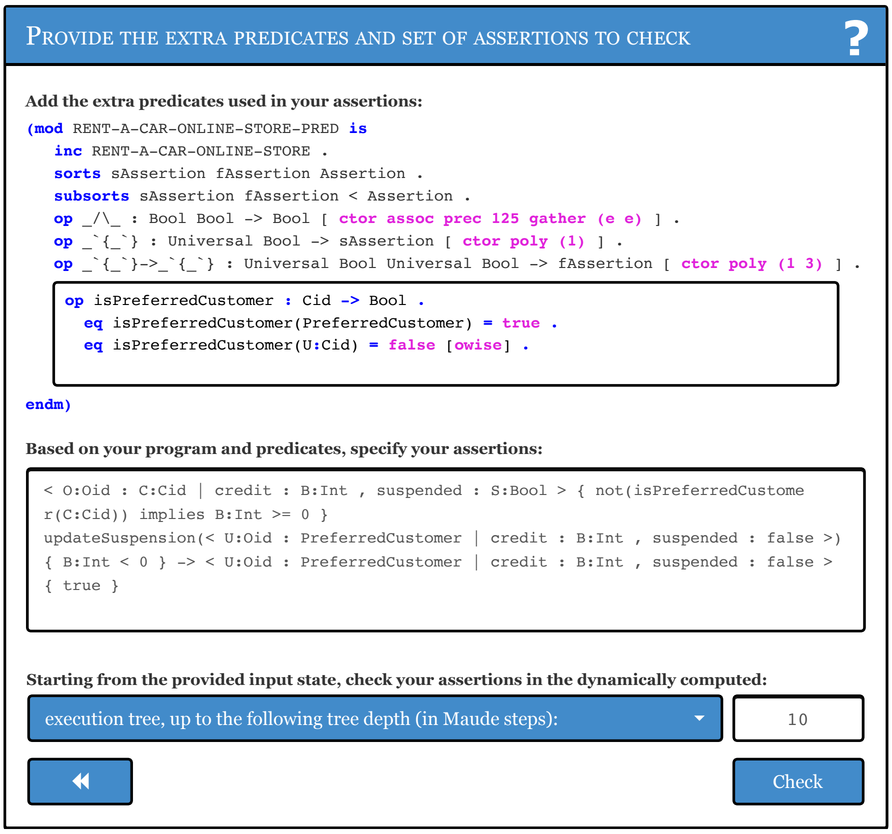
```
< 'A1 : EconomyCar | available : true , rate : 30 > < 'A3 : MidSizeCar | available :
true , rate : 45 > < 'A5 : FullSizeCar | available : true , rate : 70 > < 'C1 : Cust
omer | credit : 50 , suspended : false > < 'C2 : PreferredCustomer | credit : 100 ,
suspended : false > < 'RG : Register | rentals : 0 , date : 0 >
```

Figure 7.4: Input Phase I.

By pressing the CHECK button, the assertion checking algorithm starts and immediately discovers that Θ is not satisfied in the following Maude step $\mathcal{M}$

$s_0 \xrightarrow{\texttt{3-day-rental}}$

```
< 'A1 : EconomyCar | available : false , rate : 30 >
< 'A3 : MidSizeCar | available : true , rate : 45 >
< 'A5 : FullSizeCar | available : true , rate : 70 >
< 'C1 : Customer | credit : - 40 , suspended : false >
< 'C2 : PreferredCustomer | credit : 100 , suspended :
false >
< 'R0 : Rental | car : A1 , customer : 'C1 , deposit :
90 , dueDate : 3 , pickUpDate : 0 , rate : 30 >
< 'RG : Register | date : 0 , rentals : 1 >
```



PROVIDE THE EXTRA PREDICATES AND SET OF ASSERTIONS TO CHECK

Add the extra predicates used in your assertions:

```
(mod RENT-A-CAR-ONLINE-STORE-PRED is
   inc RENT-A-CAR-ONLINE-STORE .
   sorts sAssertion fAssertion Assertion .
   subsorts sAssertion fAssertion < Assertion .
   op _/\_ : Bool Bool -> Bool [ ctor assoc prec 125 gather (e e) ] .
   op _`{_`} : Universal Bool -> sAssertion [ ctor poly (1) ] .
   op _`{_`}->_`{_`} : Universal Bool Universal Bool -> fAssertion [ ctor poly (1 3) ] .
    op isPreferredCustomer : Cid -> Bool .
      eq isPreferredCustomer(PreferredCustomer) = true .
      eq isPreferredCustomer(U:Cid) = false [owise] .
endm)
```

Based on your program and predicates, specify your assertions:

```
< O:Oid : C:Cid | credit : B:Int , suspended : S:Bool > { not(isPreferredCustome
r(C:Cid)) implies B:Int >= 0 }
updateSuspension(< U:Oid : PreferredCustomer | credit : B:Int , suspended : false >)
{ B:Int < 0 } -> < U:Oid : PreferredCustomer | credit : B:Int , suspended : false >
{ true }
```

Starting from the provided input state, check your assertions in the dynamically computed:

execution tree, up to the following tree depth (in Maude steps):
10
Check

Figure 7.5: Input Phase II.

since `'C1`'s credit becomes negative after the application of the `3-day-rental` rule in s_0 that allows `'C1` to rent car `'A1`. Then, a system error symptom is automatically computed by the tool, which unambiguously signals the anomalous subterm

```
< 'C1 : Customer | credit : - 40 , suspended : false >
```

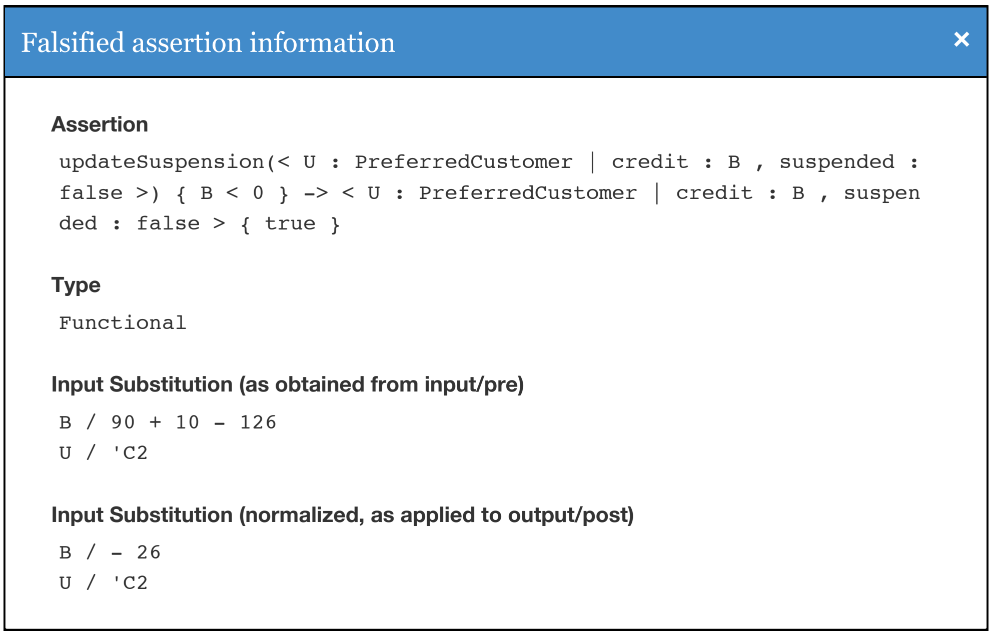
of the last state of $\mathcal{M}$, and produces the associated term slice

$$\mathtt{l}^\bullet = \bullet_1\ \bullet_2\ \bullet_3\ \texttt{<}\ \bullet_4\ \texttt{:}\ \bullet_5\ \texttt{| credit : - 40 ,}\ \bullet_6\ \texttt{>}\ \bullet_7\ \bullet_8\ \bullet_9$$

Finally, the algorithm automatically generates the backward trace slice of $\mathcal{M}$ with respect to $\mathtt{l}^\bullet$, that is,

$$\begin{array}{l}\texttt{<}\ \bullet_{10}\ \texttt{:}\ \bullet_{11}\ \texttt{| available : true , rate : 30 >}\ \bullet_{15}\ \bullet_{14}\ \texttt{<}\ \bullet_4\ \texttt{:}\ \bullet_5 \\ \quad \texttt{| credit : 50 , suspended : false >}\ \bullet_{13}\ \bullet_{12} \\ \bullet\!\!\to \\ \bullet_1\ \bullet_2\ \bullet_3\ \texttt{<}\ \bullet_4\ \texttt{:}\ \bullet_5\ \texttt{| credit : - 40 ,}\ \bullet_6\ \texttt{>}\ \bullet_7\ \bullet_8\ \bullet_9\end{array}$$

which suggests an erroneous implementation of the `3-day-rental` rule. Indeed, `3-day-rental` authorizes any car rental to all customers, even when the requested deposit exceeds the residual customer credit, which contradicts the property asserted by the system constraint Θ.


Falsified assertion information ×

Assertion

```
updateSuspension(< U : PreferredCustomer | credit : B , suspended :
false >) { B < 0 } -> < U : PreferredCustomer | credit : B , suspen
ded : false > { true }
```

Type

```
Functional
```

Input Substitution (as obtained from input/pre)

```
B / 90 + 10 - 126
U / 'C2
```

Input Substitution (normalized, as applied to output/post)

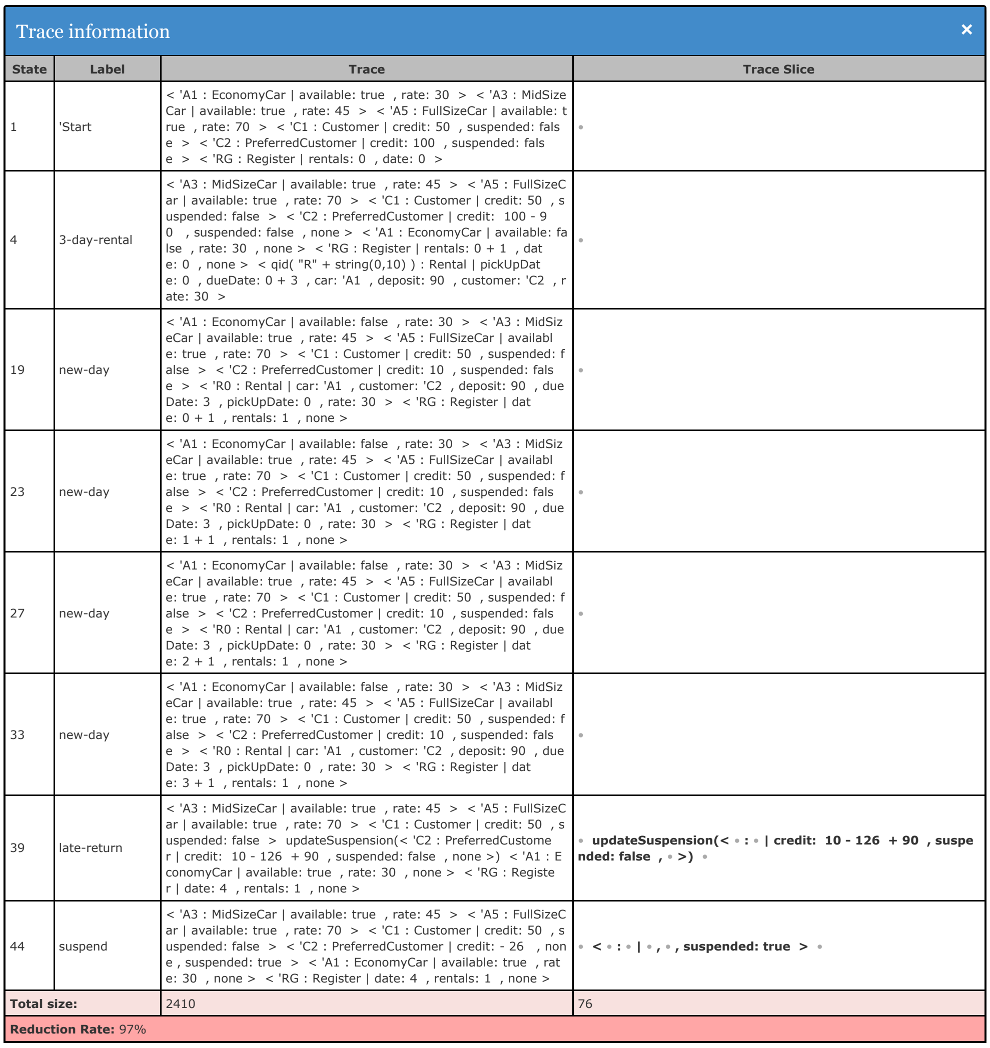
```
B / - 26
U / 'C2
```

Figure 7.6: Description of the falsified assertion Φ.

If we re-execute the analysis after correcting the buggy `3-day-rental` rule in the `RENT-A-CAR` module, we can also discover a violation of the functional assertion Φ that detects an anomalous behavior of function `updateSuspension`: in fact, `updateSuspension` suspends *every* customer with debts (i.e., a negative credit), while preferred customers should never be suspended. Details of the refuted assertion are attained by selecting the *falsified assertion* option from the tool menu, as shown in Figure 7.6.

The delivered trace slice is shown in Figure 7.7, which displays a tabular view of the trace (also provided by our tool), where the achieved reduction is shown (97%). In order to simplify the displayed view of the trace, we note that subindices of •-variables are hidden in our implementation; they can be shown by selecting the *detailed trace* view. After the diagnosis, runtime checks can be turned off to avoid any execution overheads.

Finally, by running the *program slice* option of ABETS, all program statements that can (potentially) cause the erroneous program behavior are automatically identified (see Figure 7.8).

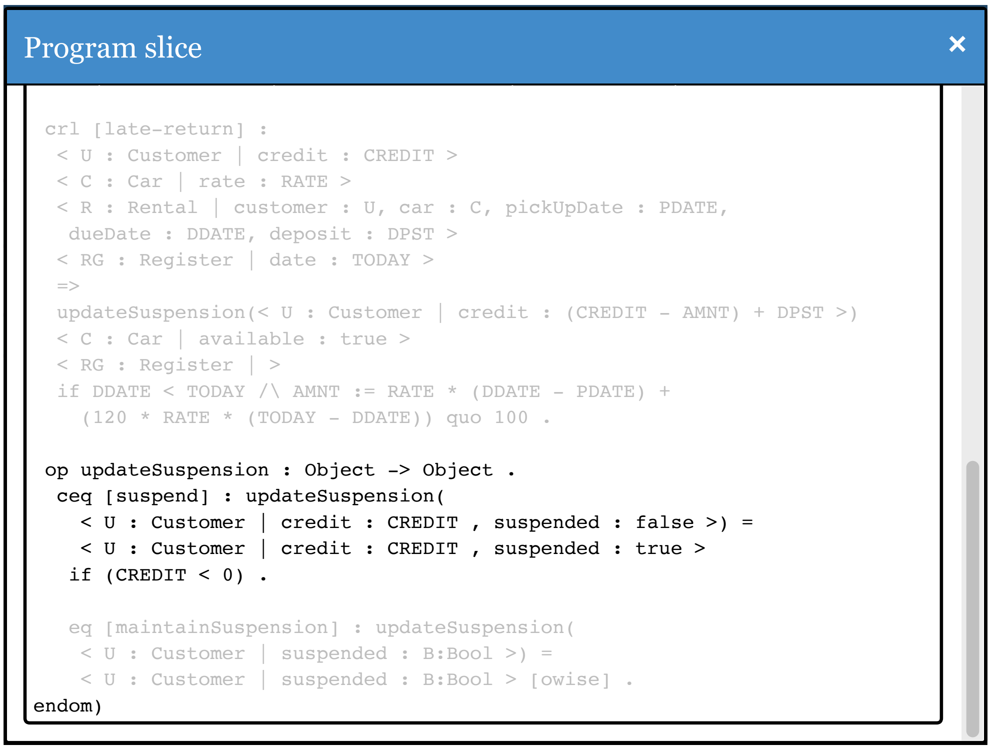

Trace information ×

State	Label	Trace	Trace Slice
1	'Start	< 'A1 : EconomyCar | available: true , rate: 30 > < 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: 100 , suspended: false > < 'RG : Register | rentals: 0 , date: 0 >	•
4	3-day-rental	< 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: 100 - 90 , suspended: false , none > < 'A1 : EconomyCar | available: false , rate: 30 , none > < 'RG : Register | rentals: 0 + 1 , date: 0 , none > < qid("R" + string(0,10)) : Rental | pickUpDate: 0 , dueDate: 0 + 3 , car: 'A1 , deposit: 90 , customer: 'C2 , rate: 30 >	•
19	new-day	< 'A1 : EconomyCar | available: false , rate: 30 > < 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: 10 , suspended: false > < 'R0 : Rental | car: 'A1 , customer: 'C2 , deposit: 90 , dueDate: 3 , pickUpDate: 0 , rate: 30 > < 'RG : Register | date: 0 + 1 , rentals: 1 , none >	•
23	new-day	< 'A1 : EconomyCar | available: false , rate: 30 > < 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: 10 , suspended: false > < 'R0 : Rental | car: 'A1 , customer: 'C2 , deposit: 90 , dueDate: 3 , pickUpDate: 0 , rate: 30 > < 'RG : Register | date: 1 + 1 , rentals: 1 , none >	•
27	new-day	< 'A1 : EconomyCar | available: false , rate: 30 > < 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: 10 , suspended: false > < 'R0 : Rental | car: 'A1 , customer: 'C2 , deposit: 90 , dueDate: 3 , pickUpDate: 0 , rate: 30 > < 'RG : Register | date: 2 + 1 , rentals: 1 , none >	•
33	new-day	< 'A1 : EconomyCar | available: false , rate: 30 > < 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: 10 , suspended: false > < 'R0 : Rental | car: 'A1 , customer: 'C2 , deposit: 90 , dueDate: 3 , pickUpDate: 0 , rate: 30 > < 'RG : Register | date: 3 + 1 , rentals: 1 , none >	•
39	late-return	< 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > updateSuspension(< 'C2 : PreferredCustomer | credit: 10 - 126 + 90 , suspended: false , none >) < 'A1 : EconomyCar | available: true , rate: 30 , none > < 'RG : Register | date: 4 , rentals: 1 , none >	• **updateSuspension(< • : • | credit: 10 - 126 + 90 , suspended: false , • >)** •
44	suspend	< 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: - 26 , none , suspended: true > < 'A1 : EconomyCar | available: true , rate: 30 , none > < 'RG : Register | date: 4 , rentals: 1 , none >	• **< • : • | • , • , suspended: true >** •
Total size:		2410	76
Reduction Rate: 97%			

Figure 7.7: Computed Trace Slice after refuting the functional assertion Φ.

7.3 Implementation Details

The ABETS tool contains about 3500 lines of Maude code, 1000 lines of C++ code, 1000 lines of Java code, and 3000 lines of JavaScript code. In the following some relevant details about the system implementation, including both the architecture of ABETS and its most relevant features are described.



```
crl [late-return] :
 < U : Customer | credit : CREDIT >
 < C : Car | rate : RATE >
 < R : Rental | customer : U, car : C, pickUpDate : PDATE,
  dueDate : DDATE, deposit : DPST >
 < RG : Register | date : TODAY >
 =>
 updateSuspension(< U : Customer | credit : (CREDIT - AMNT) + DPST >)
 < C : Car | available : true >
 < RG : Register | >
 if DDATE < TODAY /\ AMNT := RATE * (DDATE - PDATE) +
   (120 * RATE * (TODAY - DDATE)) quo 100 .

op updateSuspension : Object -> Object .
 ceq [suspend] : updateSuspension(
   < U : Customer | credit : CREDIT , suspended : false >) =
   < U : Customer | credit : CREDIT , suspended : true >
  if (CREDIT < 0) .

  eq [maintainSuspension] : updateSuspension(
   < U : Customer | suspended : B:Bool >) =
   < U : Customer | suspended : B:Bool > [owise] .
endom)
```

Figure 7.8: Computed Program Slice.

7.3.1 The System Architecture

The architecture of ABETS is depicted in Figure 7.9 and consists of the following components:

(i) a Maude-based slicer and constraint-checker core that consists of about 400 Maude function definitions (approximately 3500 lines of source code) that can run at both Core Maude and Full Maude levels interchangeably;

(ii) a scalable, high-performance NoSQL database powered by MongoDB that endows the tool with *memoization* capabilities in order to improve the response time for complex and recurrent executions;

(iii) a RESTful Web service written in Java that is executed by means of the Jersey JAX-RS API;

(iv) an intuitive user interface that is based on AJAX technology and written in HTML5 canvas and Javascript.

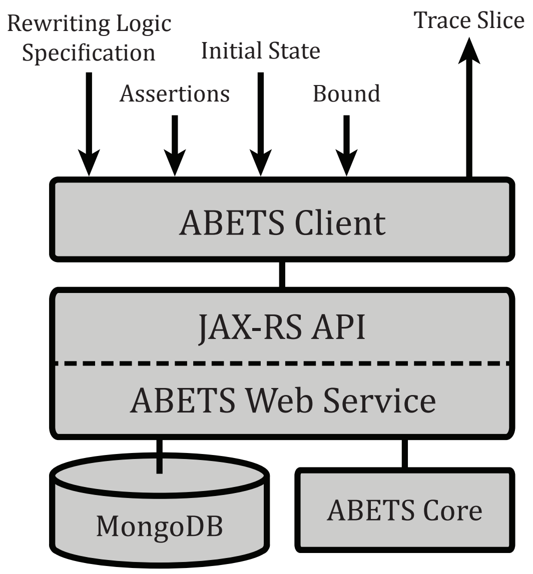


Figure 7.9: ABETS architecture.

7.3.2 Optimizations of the System

The system has been implemented by primarily focusing on its performance, including improvements for both the analysis and for the input and output operations. This section explains some relevant optimizations of the ABETS implementation and their impact on the performance of the system.

Analysis Optimizations

As already mentioned, one of the many features of ABETS is its ability to manipulate all the relevant information regarding the application of equations, algebraic axioms, and built-in operators at the meta-level, which is a feature that is not supported by Maude. In this regard, a new developer version of the Maude system called Mau-Dev has been implemented, which extends the capabilities of the latest distribution of Maude without affecting its efficiency. Moreover, to boost the system performance, the functions that are more frequently used in ABETS have been reimplemented in C++ as new, highly efficient, built-in Mau-Dev (meta-level) operations. All these new extensions are available at Mau-Dev's website at `http://safe-tools.dsic.upv.es/maudev` and explained in Chapter 8 of this thesis.

I/O Optimizations

Maude's efficient parser allows very large initial calls to be efficiently parsed in just a few milliseconds. In contrast, Full Maude's parser is entirely developed in Maude itself; hence, its efficiency can be seriously penalized when dealing with mixfix operator definitions due to extensive backtracking. As a result, ABETS initial calls that contain large and complex execution traces as arguments typically took some minutes to be loaded into previous versions of the system [Alpuente et al., 2016a].

To overcome this drawback, one can dynamically create a devoted module that defines unique *placeholder* terms that are subsequently reduced to the actual arguments of the initial (Full Maude) call. For example, to encode a Full-Maude, source-level representation of the state s_2 of Example 16, ABETS defines the 0-ary operator `abState`:

```
op abState : -> String .
eq abState = "< 'A1 : EconomyCar | available : false , rate : 30 >
    < 'A5 : FullSizeCar | available : false , rate : 70 >
    < 'C1 : Customer | credit : - 160 , suspended : false >
    < 'C2 : PreferredCustomer | credit : 10 , suspended : false >
    < 'R0 : Rental | car : 'A1 , customer : 'C2 , deposit : 90 ,
      dueDate : 3 , pickUpDate : 0 , rate : 30 >
    < 'R1 : Rental | car : 'A5 , customer : 'C1 , deposit : 210 ,
      dueDate : 3 , pickUpDate : 0 , rate : 70 >
    < 'RG : Register | date : 0 , rentals : 2 >" .
```

This greatly reduces the size of the initial Full-Maude call since it only contains the `abState` placeholder but not the actual state data. These data are later brought back by applying the `abState` equation. A similar encoding is used for user-defined assertions and execution traces that are to be asynchronously checked.

The added module is loaded prior to starting the Full Maude's *execution loop* [Clavel et al., 2007]. Thus, by taking advantage of the ability of Full Maude to access previously loaded Maude modules, the entire call can be parsed directly in Maude, except for its top-most operator.

The output of ABETS executions typically consists of a Maude term of sort `String`, represented in JSON (JavaScript Object Notation) format, that collects all the computed information (e.g., the source-level and meta-level representation of the original trace and the trace slice, the associated program slice that can be computed as described in [Alpuente et al., 2016a], and transition information between subsequent trace states). This output string is later processed by the ABETS front-end to offer a more friendly, visual representation. Since efficient output handling is crucial not to penalize the overall performance of the system, (meta) string conversion of the computed output has also been implemented in C++ as part of the Mau-Dev distribution.

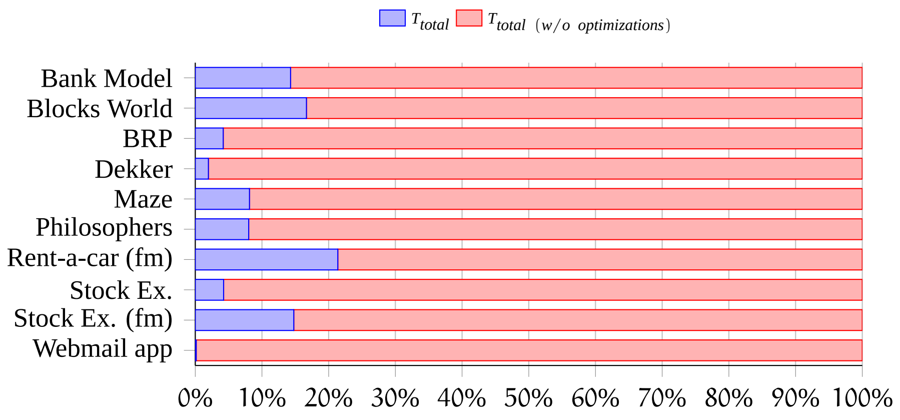


Figure 7.10: Total speedup of ABETS after optimizations.

The total speedups achieved with respect to preliminary implementations (including checking, slicing, and I/O costs) are represented in Figure 7.10, with an average speedup of 9.66 with respect to [Alpuente et al., 2016a]. The experiments that highlight the efficiency gain of the optimized system are shown in detail in Section 7.4.

7.4 Experimental Evaluation

To evaluate the performance of ABETS, the system has been benchmarked on the following collection of (Core and Full) Maude programs, which are all available within the ABETS Web platform (each program has been coupled with a suitable assertional specification, which is also available at the system's website):

- *Bank model*, a conditional Maude specification that models a faulty, distributed banking system.
- *Blocks World*, the typical AI planning problem, which consists of producing one or more vertical stacks of blocks (placed on a table) that can be moved by means of a robot arm.
- *BRP*, the Bounded Retransmission Protocol (BRP) [Helmink et al., 1994], which is a data link protocol developed and used by Philips Electronics that can be thought of as a variant of the alternating bit protocol.
- *Dekker*, a Maude specification that models a faulty version of Dekker's algorithm, one of the earliest solutions to the mutual exclusion problem which appeared in [Clavel et al., 2003].

- *Maze*, a non-deterministic Maude specification that defines a maze game in which multiple players must reach a given exit point by walking or jumping, where colliding players are eliminated from the game [Alpuente et al., 2015a].
- *Philosophers*, the classical Dijkstra dining philosophers concurrency example that deals with resource access synchronization.
- *Rent-a-car (fm)*, a *Full Maude* object-oriented system that models the logic of the faulty, distributed, object-oriented, online car-rental store of Section 4.1.2.
- *Stock Exchange*, a rewrite theory that specifies the simplified stock exchange concurrent system of Section 4.1.1.
- *Stock Exchange (fm)*, a *Full Maude*, object-oriented version of the *Stock Exchange* example.
- *Webmail*, a Maude specification borrowed from [Alpuente et al., 2014c] that models a webmail application that provides typical login/logout functionality, email management, system administration capabilities, etc.

In the experiments, both the effectiveness and the performance of ABETS have been evaluated by (synchronously) checking each program against an assertional specification that contains at least one failing assertion. This way, an erroneous execution trace $\mathcal{T}_\varepsilon$ is delivered and subsequently simplified into a trace slice $\mathcal{T}_\varepsilon^\bullet$ with regard to slicing criteria that are automatically inferred. The experiments were conducted on a PC with 3.3GHz Intel Xeon E5-1660 CPU with 64GB RAM by applying the following *modus operandi*.

1. For each program, by using the `metaRewrite` operation, Maude was forced to generate four execution traces of increasing length ($k = 10, 50, 100, 500$ rewrite steps) using its internal, default rewrite strategy, and the corresponding computation times were recorded.
2. For each execution trace, the number of assertion checks performed by the ABETS assertion-checking engine when synchronously checking the assertional specification of the corresponding program (the column `checks` in Table 7.1) were also recorded.
3. The average `slowdown` (in seconds) of five independent measurements of the execution time (in seconds) required for Point 2 with respect to Maude's computation times of Point 1 was computed.

Obviously, the slowdown of the entire checking process depends on the number of assertions that are contained in the specification and particularly on the degree of instantiation of their associated patterns. Patterns that are too general can result in a large number of (often) unprofitable evaluations of the logic formulas involved since the number of possible matchings (modulo axioms) with the system's states can grow quickly. The slowdown can also be affected by the complexity of the predicates involved in the functional and system assertions to be checked.

Table 7.1 summarizes our results. The T_{Ex} and T_{ExChk} columns measure the execution times (in ms) with and without assertion checking for traces that apply 500 rewrite rules, which expands to 8292 rewrites (i.e., rule, equation, built-in operator, and axiom applications) on average. *#Chk* represents the total number of assertion checks performed when assertion checking was enabled. *OV* is the overhead, i.e., the ratio $= (T_{ExChk} - T_{Ex})/T_{Ex}$ which indicates the relative slowdown due to assertion-checking. The results obtained are quite satisfactory and comparable with similar logic assertion checking frameworks such as [Mera et al., 2009]. The average overhead is 1.92, which is 69% of the average value (2.78) of the overhead of [Alpuente et al., 2016a] that are shown in column $OV_{\mathtt{jlamp}}$ for the very same benchmark programs.

The figures in the $\mathsf{T}_{\mathtt{synth}}$ and $\mathsf{T}_{\mathtt{fix}}$ columns respectively measure the times for synthesizing the slicing criterion and for inferring the program repairs (in ms). Our experiments show very small synthesis times for the slicing criteria that grow linearly with the size of the erroneous state. This is particularly evident in the case of Webmail App, whose states are quite large (about 2.5Kb, which is 20 times the size of the Stock Ex. states). The time for inferring the repairs is also a small portion of the total execution time.

The trace slices that are automatically delivered by ABETS are evaluated by comparing the size of the detected erroneous execution trace $\mathcal{T}_{\varepsilon}$ (in kilobytes); the size of the sliced execution trace $\mathcal{T}_{\varepsilon}^{\bullet}$ (in kilobytes); and the derived *reduction* rate achieved (*%Red.*), which ranges from 98% to 62% with an average reduction rate of 85%. With regard to the time required to perform the slicing, our implementation is quite time efficient despite the complex analyses and reasoning modulo axioms performed underneath; the elapsed times are small even for very complex traces and also scale linearly.

Finally, the generation, parsing, and input/output of traces (and trace slices) have been greatly improved in the current version of ABETS. The input/output (I/O) times are shown in column $\mathsf{T}_{I/O}$ of Table 7.1 (in ms) for I/O data sizes that range from 15 Kb (in the case of the `Blocks` program) to 7 Mb (in the case of the `Stock Ex.(fm)` program). This gives an average I/O cost of 0.6 s, whereas in our previous tool the I/O operations took minutes.

Program	T_{Ex}	T_{ExChk}	*#Chk*	*OV*	$OV_{\mathtt{jlamp}}$	T_{synth}	T_{fix}	$\mathsf{T}_{I/O}$	*Size* $\mathcal{T}_{\varepsilon}$	*Size* $\mathcal{T}_{\varepsilon}^{\bullet}$	*% Red.*
Bank Model	17	101	2,004	4.94	5.76	2	2	10	9.536	1.236	87%
Blocks World	19	37	509	0.95	2.16	1	1	2	0.279	0.046	84%
BRP	5	23	1,002	3.6	4.6	1	2	9	0.792	0.269	67%
Dekker	40	98	1,002	1.45	2.5	2	14	55	8.268	0.286	97%
Maze	128	409	7,437	2.2	3	1	3	13	2.747	0.423	85%
Philosophers	12	36	811	2	2.92	1	3	47	5.244	1.99	62%
Rent-a-car (fm)	178	263	1,503	0.48	0.52	5	9	247	5.507	0.115	98%
Stock Ex.	36	103	1,503	1.86	2.58	3	12	263	46.423	4.153	91%
Stock Ex. (fm)	726	1,310	2,004	0.8	1.72	5	43	4688	195.397	20.862	89%
Webmail app	138	271	1,002	0.96	1.99	9	20	541	133.46	7.823	94%

Table 7.1: Synchronous assertion-checking performance analysis

Chapter 8

Mau-Dev: A Developer Extension of Maude

One of the pillars on which the implementation of Maude is based is its everlasting search for extreme performance, which becomes evident from the first time one inspects the source code of Maude. This focus on efficiency has its obvious advantages when running programs or performing program analysis, but it comes at the expense of losing program execution details that, at some point, are considered expendable and thus discarded. However, information that is usually considered less relevant when performing some classical reachability-based analyses may be of extreme importance for other specialized analyses that involve the deep inspection of traces (e.g., for program debugging). For instance, the equational and axiomatic simplifications performed by Maude are not accessible at the meta level and can only be accessed as a mere textual annotation to the output of the default Maude debugger. Mau-Dev [Mau-Dev, 2016] is a developer extension of Maude that endows the official Maude distribution with new, useful developer (meta-level) operations that aim to overcome this drawback without sacrificing Maude's high performance.

This chapter presents Mau-Dev, which is a developer extension of Maude that endows the official Maude release with new meta-level operations that are natively implemented in C++ and are useful for the development of Maude applications. The chapter is organized as follows. Section 8.1 summarizes the novel native (meta) operations provided by Mau-Dev together with some examples. Section 8.2 gives some remarks on the compatibility of Mau-Dev with respect to the official Maude distribution. Finally, Section 8.3 presents some experiments that benchmark two complex yet very efficient operations available in Mau-Dev that are key for many sophisticated applications.

8.1 Novel Operations

This section summarizes the novel meta-operations provided by Mau-Dev together with some illustrative examples.

8.1.1 metaReducePath

One of the main challenges in the implementation of a Maude tool that manipulates computation traces is to make explicit the concrete sequence of internal term transformations occurring in the trace, which is generally hidden and inaccessible within Maude's rewrite machinery. For the case of rule applications, this sequence can be easily retrieved by means of the Maude `metaSearchPath` operation, but a similar operation does not exist for the sequence of built-in operators and equations applied. These are only recorded in a raw text output trace, which cannot be manipulated as a meta-level expression by Maude. In order to fill this gap, Mau-Dev supports a new meta-operation named `metaReducePath`, which returns the detailed sequence of transformations (by using equations and built-in operators) applied to a term until its canonical form is reached.

Specifically, given a Maude module `M`, a term `t`, and a Boolean expression `b`, the `metaReducePath` operation delivers an instrumented trace of sort `ITrace` that contains the precise equational simplification sequence for `t` in `M` augmented with the computed substitutions and contexts. For the case when `b` is `true`, the applied membership axioms are also included in the trace.

The formal definition of `metaReducePath`, which is part of the prelude of Mau-Dev, is as follows.

```
sorts ITraceStep ITrace ITrace? .
subsort ITraceStep < ITrace < ITrace? .

op {_,_,_} : Equation Substitution Context -> ITraceStep
                                    [ctor format(n n d n d n n n)] .
op {_,_,_} : MembAx Substitution Context -> ITraceStep
                                    [ctor format(n n d n d n n n)] .
op nil : -> ITrace? [ctor] .
op __ : ITrace? ITrace? -> ITrace?
                               [ctor assoc id: nil format (d d d)] .

op metaReducePath : Module Term Bool -> ITrace? [special (...)] .
```

Example 20. *Consider the following Maude specification from [Clavel et al., 2016] that computes the Fibonacci sequence for the input term* `fibo(2)`.

```
    fmod FIBONACCI is
      protecting NAT .

      op fibo : Nat -> Nat .
      var N : Nat .
      eq [EQ1] : fibo(0) = 0 .
      eq [EQ2] : fibo(1) = 1 .
      eq [EQ3] : fibo(s s N) = fibo(N) + fibo(s N) .
    endm
```

The simplification chain that leads to the result `1` *(i.e., the canonical form of* `fibo(2)`*), is delivered as follows:*

```
reduce in META-LEVEL : metaReducePath(upModule('FIBONACCI,false),
                                      'fibo['s_^2['0.Zero]],false) .
result ITrace:
{
  eq 'fibo['s_^2['N:Nat]] = '_+_['fibo['N:Nat],'fibo['s_['N:Nat]]]
  [label('EQ3)] .,
  'N:Nat <- '0.Zero,
  []
}
{
  eq 'fibo['0.Zero] = '0.Zero [label('EQ1)] .,
  (none).Substitution,
  '_+_[[],'fibo['s_['0.Zero]]]
}
{
  eq 'fibo['s_['0.Zero]] = 's_['0.Zero] [label('EQ2)] .
  (none).Substitution,
  '_+_['0.Zero,[]]
}
{
  eq '_+_['0.Zero,'s_['0.Zero]] = 's_['0.Zero] [none] .
  (none).Substitution,
  []
}
```

Note that the above outcome consists of a simplification trace of four steps, where the first three steps apply, in this order, the equations labelled `EQ3`*,* `EQ1`*, and* `EQ2`*, whereas the last step simplifies the resulting term by using the built-in operation* `_+_` *that computes the summation of* `0` *and* `1`*.*

8.1.2 metaGetVariantExt and metaGetIrredundantVariantExt

For the case of variant narrowing computation traces, the basic information that is necessary to visually deploy the variant narrowing trees can be essentially obtained by invoking the new meta-operations of Maude, namely `metaGetVariant` and `metaGetIrredundantVariant`. That is the only way to retrieve the precise information that makes the structure of the tree explicit. Specifically, what Maude outputs is the following (in this order): (i) the computed variant term, (ii) the computed variant substitution, (iii) the largest index n of any fresh variable appearing in the solutions, (iv) the identifier of the parent variant, and (v) a Boolean flag that indicates whether or not there are more variants in the current tree level.

Mau-Dev provides two new meta-operations, namely `metaGetVariantExt` and `metaGetIrredundantVariantExt`, which further extend the corresponding Maude meta-operations by delivering a much more informed version of the variant narrowing tree. Specifically, the new operations augment the standard output with: (i) the context in which the unification of the narrowing step has taken place, (ii) the variant equation used for the unification, (iii) the equational unifier, and (iv) the substitution that binds the computed variant with the variables of its antecedent in the narrowing tree.

Formally, the two new meta-operations and the constructor operator for the new `VariantExt` sort are defined as follows:

```
sort Equation? .
subsort Equation < Equation? .

sort VariantExt .
subsort Variant < VariantExt < Variant? .

op {_,_,_,_,_,_,_,_,_} : Term Substitution Nat Parent Bool Context
                                 Equation? Substitution Substitution
                                               -> VariantExt [ctor] .

op {_,_,_} : Equation Substitution Context -> ITraceStep
                                     [ctor format(n n d n d n n n)] .
op {_,_,_} : MembAx Substitution Context -> ITraceStep
                                     [ctor format(n n d n d n n n)] .

op noEquation : -> Equation? [ctor] .

op metaGetVariantExt : Module Term TermList Nat Nat ~> Variant?
                                                   [special (...)] .
```

```
op metaGetIrredundantVariantExt : Module Term TermList Nat Nat ~>
                                          Variant? [special (...)] .
```

Example 21. *Consider the following equational theory from [Clavel et al., 2016] for the exclusive-or symbol* `_*_`[1]*, where* `mt` *is the (empty set) identity element. Note that the notation* `[NatSet]` *denotes the kind of sort* `NatSet` *that, in addition to normal data of sort* `NatSet`*, can also contain "error expressions".*

```
    fmod EXCLUSIVE-OR-NOFVP is

      sorts Nat NatSet .
      subsort Nat < NatSet .

      op s : Nat -> Nat .
      ops 0 mt : -> Nat .
      op _*_ : NatSet NatSet -> NatSet [assoc comm] .

      var X : Nat .
      var Z : [NatSet] .

      eq [idem] : X * X = mt [variant] .
      eq [idem-Coh] : X * X * Z = Z [variant] .
      eq [id] : X * mt = X [variant] .
    endm
```

The execution of the default `metaGetVariant` *meta-operation of Maude asking for the first possible variant (i.e., solution number zero) of the term* `X:[NatSet]* Y:[NatSet]` *yields the following result:*

```
reduce in META-LEVEL : metaGetVariant(upModule('EXCLUSIVE-OR-NOFVP,
                        false),'_*_[ 'X:`[NatSet`], 'Y:`[NatSet`]],
                                                     empty, 0, 0) .

result Variant: {'_*_[ '#1:`[NatSet`], '#2:`[NatSet`]],
                 'X:`[NatSet`] <- '#1:`[NatSet`] ;
                 'Y:`[NatSet`] <- '#2:`[NatSet`],2,none,false}
```

whereas the extended `metaGetVariantExt` *meta-operation of Mau-Dev delivers the following augmented outcome:*

[1]An exclusive union operator `_*_` for sets of natural numbers, `NatSet`, such that `X1* X2` is the set of natural numbers appearing in `X1` or `X2`, but not both.

```
reduce in META-LEVEL :
             metaGetVariantExt(upModule('EXCLUSIVE-OR-NOFVP,false),
                  '_*_['X:`[NatSet`],'Y:`[NatSet`]], empty, 0, 0) .

result VariantExt: {'_*_['#1:`[NatSet`],'#2:`[NatSet`]],
                    'X:`[NatSet`] <- '#1:`[NatSet`] ;
                    'Y:`[NatSet`] <- '#2:`[NatSet`],2,none,false,[],
                    noEquation,none,none}
```

Note that the first possible variant always consists of a simplified, renamed version of the input term, hence there is no actual narrowing step with additional information. Nevertheless, if we ask for the variant number five, we originally get the following result:

```
reduce in META-LEVEL : metaGetVariant(upModule('EXCLUSIVE-OR-NOFVP,
                          false),'_*_['X:`[NatSet`],'Y:`[NatSet`]],
                                                       empty, 0, 5) .

result Variant: {'%2:`[NatSet`],
                 'X:`[NatSet`] <- '_*_['%1:Nat,'%2:`[NatSet`]] ;
                 'Y:`[NatSet`] <- '%1:Nat,2,0,true}
```

whereas the `metaGetVariantExt` *delivers:*

```
reduce in META-LEVEL :
             metaGetVariantExt(upModule('EXCLUSIVE-OR-NOFVP,false),
                  '_*_['X:`[NatSet`],'Y:`[NatSet`]], empty, 0, 5) .

result VariantExt: {'%2:`[NatSet`],
                    'X:`[NatSet`] <- '_*_['%1:Nat,'%2:`[NatSet`]] ;
                    'Y:`[NatSet`] <- '%1:Nat, 2,0,true,[],
                   eq '_*_['X:Nat,'X:Nat,'Z:`[NatSet`]] =
                      'Z:`[NatSet`] [variant label('idem-Coh)] .,
                    'X:Nat <- '%1:Nat ;
                    'Z:`[NatSet`] <- '%2:`[NatSet`],
                    '#1:`[NatSet`] <- '_*_['%1:Nat,'%2:`[NatSet`]] ;
                    '#2:`[NatSet`] <- '%1:Nat}
```

Note that a meaningful variant narrowing step has been performed, hence the precise information related to that step is fully delivered.

8.1.3 metaAssociative and metaCommutative

Given a Maude module *M* and a term *t*, `metaAssociative` returns `true` if the topmost operator of *t* is associative or `false` otherwise. This operation improves the `isAssociative` function defined in Full Maude by directly inspecting a specific flag in the C++ term representation of *t* (while `isAssociative` matches *t* with the possibly large list of operator definitions in *M*). The meta-operation `metaCommutative` is the analogous of the `metaAssociative` operation for the `comm` attribute.

Example 22. *Consider the default module* `NAT` *included in the prelude of Maude together with the summation operator* `_+_`*, which is both associative and commutative, the symmetric difference operator* `sd`*, which is commutative but not associative, and the quotient operator* `quo`*, which is neither associative nor commutative. Then, if we ask for the axiomatic properties of terms constructed by using one of the above operators as the root symbol, we get the following results in zero time.*

```
reduce in META-LEVEL : metaAssociative(upModule('NAT,false),
                              '_+_['s_^2['0.Zero],'s_^2['0.Zero]]) .
result Bool: true
-------------------------------------------------------------------
reduce in META-LEVEL : metaAssociative(upModule('NAT,false),
                               'sd['s_^2['0.Zero],'s_^2['0.Zero]]) .
result Bool: false
-------------------------------------------------------------------
reduce in META-LEVEL : metaCommutative(upModule('NAT,false),
                              '_+_['s_^2['0.Zero],'s_^2['0.Zero]]) .
result Bool: true
-------------------------------------------------------------------
reduce in META-LEVEL : metaCommutative(upModule('NAT,false),
                             '_quo_['s_^2['0.Zero],'s_^2['0.Zero]]) .
result Bool: false
```

8.1.4 metaOutermost

Given a Maude module and a term *t*, `metaOutermost` delivers the list of outermost redexes of *t*. Note that this operation requires all constructor non-reducible operators occurring in the program to be explicitly identified with the `ctor` attribute.

Example 23. *Consider the following, simple Maude specification.*

```
    mod OUTERMOST is
      protecting NAT .
      op a : -> Nat [ctor] .
      op b : -> Nat .
      op f : Nat Nat -> Nat [ctor assoc comm] .

      eq b = a .
    endm
```

Then, the outermost redex of term `f(a,a,b)` *is* `b`*, as shown below:*

```
reduce in META-LEVEL : metaOutermost(upModule('OUTERMOST,false),
                                        'f['a.Nat,'a.Nat,'b.Nat]) .
result Constant: 'b.Nat
```

Moreover, term `f(a,a)` *has no outermost redexes, since both its root operator* `f` *and constant arguments* `a` *are constructors. Hence, the execution of* `metaOutermost` *delivers the* `empty` *list as shown below:*

```
reduce in META-LEVEL : metaOutermost(upModule('OUTERMOST,false),
                                                'f['a.Nat,'a.Nat]) .
result EmptyCommaList: empty
```

Finally, for input term `f(b,b)`*, the execution of* `metaOutermost` *yields a list with the two occurrences of constant* `b`*, which indeed are the two outermost redexes of the term:*

```
reduce in META-LEVEL : metaOutermost(upModule('OUTERMOST,false),
                                                'f['b.Nat,'b.Nat]) .
result NeGroundTermList: 'b.Nat,'b.Nat
```

8.1.5 metaConstructor

Given a Maude module M and a term t, `metaConstructor` returns `true` if the top-most operator of t is a constructor symbol of M. Similarly to operation `metaOutermost`, this operation expects constructor operators to be properly identified by means of the `ctor` attribute.

Example 24. *Consider the following, simple Maude specification.*

```
  mod CONSTRUCTOR is
    protecting NAT .
    op f : Nat -> Nat [ctor] .
    op g : Nat -> Nat .
  endm
```

Then, the root operator of `f(#0:Nat)` *is correctly identified to be a constructor:*

```
reduce in META-LEVEL : metaConstructor(upModule('CONSTRUCTOR,false),
                                                    'f['#0:Nat]) .
result Bool: true
```

whereas a similar call for term `g(#0:Nat)` *returns* `false`*, since* `g` *is not a constructor operator of the given theory:*

```
reduce in META-LEVEL : metaConstructor(upModule('CONSTRUCTOR,false),
                                                    'g['#0:Nat]) .
result Bool: false
```

8.1.6 metaIdentity, metaRightIdentity, and metaLeftIdentity

Given a Maude module M and a term t, `metaIdentity` delivers the identity element of the topmost symbol of t or `noIdentity` if no such term exists. Analogously, the associated right (resp. left) identity element can be obtained by means of the `metaRightIdentity` (resp. `metaLeftIdentity`) operation. For terms with symbols obeying both left and right identity, all three operations deliver the same result.

Example 25. *Consider the following, simple Maude specification.*

```
  mod IDENTITY is
    sorts mySort mySortList .
    subsort mySort < mySortList .

    op nil : -> mySortList [ctor] .
    op _i_ : mySortList mySortList -> mySortList
                                  [ctor assoc id: nil] .
    op _l_ : mySortList mySortList -> mySortList
                             [ctor assoc left id: nil] .
    op _r_ : mySortList mySortList -> mySortList
                            [ctor assoc right id: nil] .
  endm
```

Then, we can ask for the identity element of terms constructed by using one of the above operators as the root symbol as follows:

```
reduce in META-LEVEL : metaIdentity(upModule('IDENTITY,false),
                                    '_i_['#0:mySort,'#1:mySort]) .
result Constant: 'nil.mySortList
-------------------------------------------------------------------
reduce in META-LEVEL : metaLeftIdentity(upModule('IDENTITY,false),
                                    '_i_['#0:mySort,'#1:mySort]) .
result Constant: 'nil.mySortList
-------------------------------------------------------------------
reduce in META-LEVEL : metaRightIdentity(upModule('IDENTITY,false),
                                    '_i_['#0:mySort,'#1:mySort]) .
result Constant: 'nil.mySortList
-------------------------------------------------------------------
reduce in META-LEVEL : metaIdentity(upModule('IDENTITY,false),
                                    '_l_['#0:mySort,'#1:mySort]) .
result Term?: noIdentity
-------------------------------------------------------------------
reduce in META-LEVEL : metaLeftIdentity(upModule('IDENTITY,false),
                                    '_l_['#0:mySort,'#1:mySort]) .
result Constant: 'nil.mySortList
-------------------------------------------------------------------
reduce in META-LEVEL : metaRightIdentity(upModule('IDENTITY,false),
                                    '_l_['#0:mySort,'#1:mySort]) .
result Term?: noIdentity
-------------------------------------------------------------------
reduce in META-LEVEL : metaIdentity(upModule('IDENTITY,false),
                                    '_r_['#0:mySort,'#1:mySort]) .
result Term?: noIdentity
-------------------------------------------------------------------
reduce in META-LEVEL : metaLeftIdentity(upModule('IDENTITY,false),
                                    '_r_['#0:mySort,'#1:mySort]) .
result Term?: noIdentity
-------------------------------------------------------------------
reduce in META-LEVEL : metaRightIdentity(upModule('IDENTITY,false),
                                    '_r_['#0:mySort,'#1:mySort]) .
result Constant: 'nil.mySortList
```

Note that, for the case of the mixfix operator `i`*, which has both left and right identity, all three results are the same, whereas for the* `l` *and* `r` *operators, only the corresponding call returns the identity term* `nil` *while all other calls return the constant* `noIdentity`.

8.1.7 metaString

Given a Maude module M, a term t, and a Boolean expression b, the meta-operation `metaString` returns a term of sort `String` that provides the source-level (resp. meta-level) representation of t for the case when b is true (resp. false), which highly outspeeds any possible (user-defined) Maude counterpart.

Example 26. *Consider the (non-simplified) term* `2 + 3`*. Then, if we ask for the source-level (respectively meta-level) representation of the term, we get the following results:*

```
reduce in META-LEVEL : metaString(upModule('NAT,false),
                         '_+_['s_^2['0.Zero],'s_^3['0.Zero]],true) .
result String: "2 + 3"
-------------------------------------------------------------------
reduce in META-LEVEL : metaString(upModule('NAT,false),
                        '_+_['s_^2['0.Zero],'s_^3['0.Zero]],false) .
result String: "'_+_['s_^2['0.Zero],'s_^3['0.Zero]]"
```

Note that, in the second case, the input term and the result seem to be the very same term, but they are not. Actually, the input term is of sort `Term`*, whereas the result is of sort* `String`*.*

8.1.8 metaMap

Given a Maude module M and a term t, `metaMap` delivers a sophisticated string representation of the term t where each string character in the representation has a shortcut to the corresponding subterm of t.

The returned string is map of positions, a sequence of elements of the form cNpP where c stands for chars, N is a natural number that specifies a length (of chars to be consumed from the source-level representation of the input term), p stands for position and P is either empty (meaning that the associated position is Λ), a single position (e.g., `2.1`), or a set of positions separated by the `X` char (e.g., `1X1.1`).

Example 27. *Consider again the (non-simplified) term* `2 + 3`*. Then, if we ask for the map associated with its source-level representation, we get the following result:*

```
reduce in META-LEVEL : metaMap(upModule('NAT,false),
                               '_+_['s_^2['0.Zero],'s_^3['0.Zero]]) .
result String: "c1p1X1.1c1pc1pc1pc1p2X2.1"
```

which can be explained as follows:

- *The first component of the map* c1p1X1.1c1pc1pc1pc1p2X2.1 *indicates that the first character of string "*2 + 3*" (i.e.,* 2*) is associated with positions* 1 *and* 1.1 *(*c1p1X1.1c1pc1pc1pc1p2X2.1*).*
- *The following component of map* c1p1X1.1c1pc1pc1pc1p2X2.1 *indicates that the next character of "*2 + 3*", i.e., a blank space, is associated with position* Λ *(*c1p1X1.1c1pc1pc1pc1p2X2.1*).*
- *The next component* c1p1X1.1c1pc1pc1pc1p2X2.1 *(i.e.,* + *) is also associated with position* Λ *(*c1p1X1.1c1pc1pc1pc1p2X2.1*), and so on.*

Note that, by means of this new meta-operation, developers can easily and unambiguously locate all the positions of a term in their respectively source-level representations without having access to the semantic properties of the theory.

8.2 Compatibility

Mau-Dev is fully compatible with its corresponding official Maude release and preserves its original behavior and efficiency. Moreover, Mau-Dev assumes that variables whose quoted identifier starts with "#!" (e.g., #!1:Nat) are variables of no interest (called bullets) and therefore their source-level representation will always be the token DEV-BULLET, regardless of their actual internal (meta-level) representation, which is always preserved. Note that this is just a cosmetic restriction, since bullets are internally handled as simple variables. Moreover, since "DEV-BULLET" is a special token representing the • character, its length (in chars) is considered to be 1 with respect to the metaMap operation.

Example 28. *Consider two variables, namely* #1:Nat *and* #!2:Nat*. Then,* #!2:Nat *is interpreted as a bullet because its quoted identifier starts with "#!", as shown below:*

```
Mau-Dev> red #1:Nat .
reduce in CONVERSION : #1:Nat .
result Nat: #1:Nat
-------------------------------------------------------------------
Mau-Dev> red metaReduce(upModule('NAT,false),'#1:Nat) .
reduce in META-LEVEL : metaReduce(upModule('NAT,false),'#1:Nat) .
result ResultPair: {'#1:Nat,'Nat}
--------------------------------------------------------------------
Mau-Dev> red #!2:Nat .
reduce in CONVERSION : DEV-BULLET .
```

```
result Nat: DEV-BULLET
------------------------------------------------------------------
Mau-Dev> red metaReduce(upModule('NAT,false),'#!2:Nat) .
reduce in META-LEVEL : metaReduce(upModule('NAT,false),'#!2:Nat) .
result ResultPair: {'#!2:Nat,'Nat}
```

8.3 Experimental Evaluation

This section presents the experiments performed in order to evaluate the efficiency of the two most complex meta-operations provided by Mau-Dev (i.e., the `metaReducePath` and the `metaGetVariantExt` meta-operations) and measure them against the current distribution of Maude. Note that the remaining meta-operations provided by Mau-Dev deliver their results almost instantly.

8.3.1 metaReducePath

Technically, the execution of `metaReducePath` can be split into two phases: equational simplification and lifting to the meta-level. In the simplification phase, the input term is reduced to its canonical form by using Maude's equational simplification. For each applied equation and internal normalization transformation, `metaReducePath` additionally collects all the relevant information that is needed to subsequently reconstruct the performed steps. This includes not only built-in evaluations but also *memoizations* and other internal manipulations such as `iter` transformations, which replace chains of iterations of a unary operator by a single instance of the iterated function, raised to the number of iterations, e.g., `s(s(s(0)))` as $\mathtt{s}^3\mathtt{(0)}$. Once the term has reached its canonical form, the lifting phase consists of raising to the meta-level all the collected information and assembling the resulting instrumented computation in one single piece.

Table 8.1 provides some figures regarding the execution of the new `metaReducePath` operation. The `metaReducePath` operation has been tested on a 3.3GHz Intel Xeon E5-1660 with 64GB of RAM by reducing different calls to the `fibo` function of the Fibonacci specification of Example 20. In Table 8.1, the two phases mentioned above are distinguished, namely equational simplification and lifting. For the equational simplification phase, the number of rewrites and the reduction times are provided. For the lifting phase, the problem size, the length of the resulting instrumented computation, and the processing times are shown instead. The problem size (column ♮ size) is measured as the number of expressions (applied equation, substitution, and context for each step) that are manipulated. The length of the resulting instrumented computation (column $|\mathcal{T}|$) is measured as the number of rewrite

	equational simplification		*meta-level lifting*		
n	*rewrites*	*time (s)*	$\natural$ *size*	$\lvert\mathcal{T}\rvert$	*time (s)*
5	22	0	78	26	0
10	265	0	957	319	0
15	2,959	0.02	10,704	3,568	0.04
20	32,836	0.24	118,800	39,600	0.73
25	364,177	3.41	1,317,603	439,201	10.18

Table 8.1: Execution results of the `metaReducePath` operation for `fibo(n)`.

steps. Note that for extremely huge computations such as the trace of `fibo(25)`, which consists of 439,201 rewrite steps, the number of manipulated terms can be very high (more than 1,300,000), yet the execution time is reasonable (a few seconds) and comparable to existing Maude meta-operations that process millions of terms [Eker, 2003].

Finally, it is worth mentioning that the `metaReducePath` meta-operation takes into account the *Church-Rosser* and *termination* properties of functional modules assumed by Maude. Therefore, it returns just one possible simplification sequence that perfectly reproduces the normalization carried out by Maude following its internal strategy while ignoring the rest of the alternative normalizations.

8.3.2 metaGetVariant vs metaGetVariantExt

As for the variant-narrowing related meta-operations, Table 8.2 provides some figures regarding the execution of the new `metaGetVariantExt` operation in comparison with the standard `metaGetVariant` operation. The two implementations have been tested on a 3.3GHz Intel Xeon E5-1660 with 64 GB of RAM by generating a number of variants for a collection of Maude programs: *Exclusive-or*, the classical specification of the boolean XOR; *Fibonacci* (see Example 20), a Maude specification that computes the Fibonacci sequence; *Flip-graph*, a variant of the classical *flip* function for binary graphs (instead of trees) taken from [Alpuente et al., 2016c]; and *Parser*, a generic parser for languages generated by simple, right regular grammars also from [Alpuente et al., 2016c]. Specifically, for each Maude program, Mau-Dev has computed three different numbers of variants, which takes from a few seconds to a few minutes to be generated. The `metaGetVariant` invocations have been measured on a statically compiled version of the last alpha release of Maude (alpha 113), whereas the `metaGetVariantExt` invocations have been benchmarked on a Mau-Dev executable that is based on the same alpha version.

		`metaGetVariant`		`metaGetVariantExt`	
	variants	*size (kB)*	*time (s)*	*size (kB)*	*time (s)*
Exclusive-or	40	7.37	2.49	12.34	2.48
	45	8.81	24.82	14.42	24.56
	50	10.37	302.18	16.62	299.29
Fibonacci	40	520.23	3.51	1,417.26	3.59
	45	2,198.07	20.52	5,151.39	20.94
	50	5,751.55	406.59	15,675.13	415.14
Flip-graph	500	4,804.66	3.05	7,259.92	3.09
	1,000	19,520.91	30.33	29,387.01	30.93
	2,000	80,372.41	360.29	120,769.01	361.54
Parser	2,500	1,961.51	3.91	3,067.46	3.92
	5,000	5,027.82	16.88	7,238.53	17.37
	10,000	13,178.03	81.64	17,598.87	81.99

Table 8.2: Execution results of the `metaGetVariant` operation and its `metaGetVariantExt` extension.

The two `size` columns correspond to the size (in kilobytes) of the generated narrowing tree (up to the requested variant), whereas the two `time` columns show the average of five different measures of the computation time (in seconds). As the experiments show, the incurred overheads w.r.t. the original meta-operation are almost negligible. Note that even for extremely huge narrowing trees, the amount of data handled is much higher w.r.t. the original meta-operation (with an average increasement factor of 1.8) yet the execution times are practically identical. Actually, some executions are even faster in the extended version (e.g., computing the fiftieth variant of the exclusive-or example), which can be explained by the side-effects of Maude's garbage collector and cache memory hits and misses.

Chapter 9

Conclusion

This chapter briefly describes in Section 9.1 those strands of research that have influenced this work the most. Section 9.2 summarizes the thesis results, followed by some conclusions and directions for future work in Section 9.3.

9.1 Related Work

Tools that are useful for mechanically checking that annotated programs meet their specifications fall into two main, complementary categories: runtime assertion checking (i.e., the testing of specifications during program execution, with any violation resulting in special errors being reported) and static verification (where logical techniques are used to prove, before execution time, that no violations of specifications will take place at runtime). It was by the mid '70s when researchers realized that monitoring assertions during program execution offers a simple and practical counterpart to formal proofs of correction. Assertion checking cannot prove that a program is correct but it does support a greater degree of automation than deductive verification, i.e., static verification of the assertions using a theorem prover, which furthermore requires the user to have broad mathematical skills and provide fairly precise and complete specifications [Din et al., 2014]. Runtime assertion checking does not face the same difficult challenges as, say, model-checking and theorem proving and is likely closer to becoming part of mainstream software development environments. The gist of runtime verification and its convenience as a partner of model-checking, theorem proving, and program testing is discussed in, e.g., [Leucker, 2012, Leucker and Schallhart, 2009].

Initially developed as a means of stating expected or desired program properties as a necessary step in constructing formal, deductive proofs of program correctness, the key role of assertions in software engineering applications has witnessed the growth of assertion notations, such as JML, OCL, Spec#, and Z, and assertion capabilities in widely used programming languages such as C#, C++, Eiffel, and Java (see [Clarke and Rosenblum, 2006, Burdy et al., 2005, Barnett et al., 2011] and further references therein). In general, assertions are supported in programming languages in one of two ways —either incorporating assertion constructs into the design of the language, or by using an external assertion language that is injected into the target programming language through suitable software wrappers. Assertions are embedded in the type systems of many programming languages that support strong typing via type declarations, where a type restriction on arguments can be considered a precondition. Some languages support even stronger typing and subtyping assertions (e.g., Maude's membership axioms, which are used to automatically 'narrow' the type T of a value into a subtype of T). Assertions may be used statically to support program analysis and also for secondary purposes, such as documentation and to provide information to an optimizer during code generation. The most obvious way to dynamically use assertions is to test them at runtime and report any detected violations. Yet assertions may be applied for automated error detection during any activity in which a program is executed, including debugging, testing, and operation.

Runtime verification (RV) is a light-weight formal technique that allows checking whether a *run* of a system under scrutiny satisfies or violates a given property [Leucker and Schallhart, 2009], or more precisely, a(n) (informative) *finite prefix* of a run, i.e., a finite execution trace. Common properties include state-based properties such as preconditions, normal and exceptional post-conditions, invariants, and history constraints. One prominent feature of RV is its being performed at runtime, which opens up the possibility to act whenever incorrect software behavior is detected. Its distinguishing research effort lies in synthesizing (on-line/off-line) monitors from high-level specifications, where a monitor is a device that reads a finite trace and efficiently yields a certain verdict, typically a truth value from some truth domain. Closely related, online monitors incrementally check the current execution of the system, while offline monitors work on a (finite set of) recorded execution(s). The problem of generating monitors can be compared to the generation of automata in model-checking, where it finds its origins.

The use of contracts or assertions to obtain more reliable programs has been proposed for many programming languages and paradigms. This is a field that has a great amount of related work; here we can only summarize a small part that is most closely related to this thesis. A runtime checker written in Maude for the executable modeling language ABS is described in [Din et al., 2014]. In functional programming, a semantics for dynamically checking contracts was first formalized in [Blume and McAllester, 2006]. Hybrid (mixed static/dynamic) contract checking for functional languages has received increasing research attention, as recently discussed in [Xu, 2012]. The notion of specifications and contracts for lazy functional (logic) programs is introduced in [Antoy and Hanus, 2012], where Curry is used as a single language for efficient implementations, executable specifications (describing the intended meaning of an operation as required for program verification), and contracts (run-time checked assertions consisting of both a pre- and a post-condition given as Boolean functions that can be weaker than a precise specification). In [Antoy and Hanus, 2012], post-conditions can be derived from existing program specifications in order to (hopefully) detect incorrect implementations. In contrast to our work, dynamic assertion checking is achieved by integrating the contract into the implementation, that is, all existing pre- and post-conditions are translated into correlated function conditions. Also different from our work, any result that a function produces must satisfy the function's (Boolean) post-condition, while we are able to discriminate among cases by specifying different state templates I and conditions φ_{in} in functional assertions $I\{\varphi_{in}\} \implies O\{\varphi_{out}\}$.

The Maude Formal Environment (MFE) is a recent effort to integrate and interoperate most of the available Maude analysis and verification tools [Martí-Oliet et al., 2012]. It includes several program analyzers and theorem provers, which are all accessible in [Maude-Tools, 2010]. Other available tools in [Maude-Tools, 2010] are not yet integrated, such as the declarative debugger of Maude [Riesco et al., 2012] and Maude's model checkers [Clavel et al., 2007, Bae and Meseguer, 2015]. The declarative debugger is based on the algorithmic debugging technique of [Shapiro, 1982] and supports the debugging of wrong results (erroneous reductions, sort inferences, and rewrites) and incomplete results (not completely reduced normal forms, greater than expected least sorts, and incomplete sets of reachable terms) in object-oriented, parameterized modules written in (Full) Maude. The declarative debugging process starts from a computation that is considered incorrect by the user (unexpected outcome) and tries to locate a program fragment that is responsible for that error symptom. The tool builds a debugging tree that represents the anomalous computation and guides the user while he/she explores the tree to find the bug. Moreover, the debugger offers the user several options to prune and traverse the tree. During the process, the system asks questions to an external oracle (typically the user) until a so-called buggy node is found (i.e., a node that contains an erroneous result but whose children have all correct results). Since a buggy node produces an erroneous output from correct inputs, it corresponds to an erroneous fragment of code that is pointed out as an error. Typical questions to the user have the form "Is it correct that term t rewrites (or fully reduces) to t'?" When the debugging tree is large, a main drawback is the frequency, size, and complexity of the questions to the oracle; hence, the tool allows some statements (and even whole modules) to simply be trusted in order to alleviate the process. We believe the slicing capabilities described in this thesis could be of great help to further shorten the declarative debugging process, avoiding unnecessary questions to the user while allowing her to identify the very buggy components within relatively large nodes. To the best of our knowledge, no general built-in support is provided in the MFE for runtime assertion checking or related disciplines such as *contract enforcement* in order to monitor *contract fulfillment* or enforce some penalty when a contract violation is observed.

Related to this thesis, a generic strategy is defined in [Roldán et al., 2009] to guarantee in Maude that a set of invariants (that can be expressed in different logics) are satisfied at every computed state. This is achieved by avoiding the execution of actions that otherwise would conduct the system to states that do not satisfy the constraints. This is in contrast to our approach in two ways. On the one hand, our assertions are *external* and evaluated at runtime, whereas driving the system's execution in such a way that every computation state complies with the constraints makes the assertions *internal* to the programmed strategy. On the other hand, the strategy of [Roldán et al., 2009] never results in violated assertions, which is essential in our approach for automated trace slicing to be fired. As another difference, we are able to check assertions that regard the normalizations carried out by using the equational part of the rewrite theory. In [Roldán and Durán, 2011], a dynamic validator of OCL constraints (class invariants and method pre/post-conditions) is proposed that evaluates Maude prototypes of UML models where both, UML models and OCL expressions, are represented as Maude specifications. OCL is specially tailored to specify constraints or queries over UML model objects; that is, the constraints are used to give an exact description of the information contained in the UML models and the queries are used to analyze these models and to validate them. Although OCL is not specific to any programming language, it explicitly targets UML in the same way that JML is tied to Java. In contrast, our notation is independent from the target programming language or modeling language so that, by keeping our syntax close to Maude, assertions can be easily specified by any Maude developer who wants to analyze a Maude representation of any programs or models of interest.

Finally, a parametric trace slicing and monitoring methodology is formalized in [Chen and Roşu, 2009]. This technique allows parametric execution traces (i.e., traces which contain events with parameter bindings) to be sliced and subsequently checked online with respect to parametric properties. It is worth noting that both the notion of execution trace of [Chen and Roşu, 2009] and the accompanying slicing algorithm differ from ours. In our framework, an execution trace is a Maude computation consisting of a rich combination of rule, equation, membership and axiom applications that is sliced by tracking backwards the relevant symbols of an automatically synthesized slicing criterion, whereas [Chen and Roşu, 2009] defines traces as sequences of parametric events that are distributed into the corresponding trace slices by analyzing their associated parameter values.

9.2 Discussion of Results

Checking assertions is a popular approach to program error discovery, debugging, and optimization. In this thesis, we have formalized a framework that integrates dynamic slicing and runtime assertion checking to help diagnose programming errors in rewrite theories and to suggest possible program repairs that remove erroneous computations.

The proposed methodology smoothly blends in with the general slicing framework of [Alpuente et al., 2014a] and the framework for the analysis and exploration of rewriting logic computations of [Alpuente et al., 2015a]. The main improvement obtained is that no error symptom must be separately identified because the assertions (or more precisely, their runtime checks) are used to synthesize deft slicing criteria. In other words, false assertions not only flag error symptoms, but, more importantly, they are used as the starting point for automated backward slicing transformation.

Assertions are usually specified either manually by the developers or automatically as a result of requirement or code analysis techniques. In our framework, (system and functional) assertions are constructed by using a simple language based on logic formulas and state patterns, which are both encoded in the same language as the target program. Hence, manual specification of assertions requires no additional learning effort to the developer, since it is assumed that he/she has is skilled at that language, and the possible intricacy of encoding automatically generated assertions is in line with the language complexity.

The proposed framework has been implemented in the ABETS prototype tool, which provides a highly dynamic environment for the runtime assertion-checking of rewriting logic theories. Our preliminary experience has shown that the synergistic capabilities of ABETS can provide a very powerful Swiss Army knife in error diagnosis and debugging by abetting the analyst's attention to suspicious (but otherwise possibly overlooked) aspects of the code.

The techniques we have developed are adequately fast and usable, with a performance that is comparable to Maude itself when applied to programs of several hundred lines. Yet, we have improved our prototype implementation in several ways. For instance, in order to access undisclosed critical information concerning equational and axiomatic simplifications in a controlled manner, ABETS relies on a custom version of Maude, called Mau-Dev, which offers a new, fast native C++ implementation of several commonly used Maude operations, thereby providing remarkable time optimizations to our developed tools. We have also included a facility to refine the inferred slicing criteria by enhancing the processing of postconditions to reduce the number of variables that are worth observing and also to add further flexibility to the selection of the violated assertion(s) to consider.

Our methodology can be straightforwardly adapted to infer slicing criteria for all failing assertions. However, from our own experience we find it is overwhelming for the user to receive all (alternative) criteria together at once. Furthermore, once an erroneous state is discovered, the execution traces that originate from it become *tainted* in the sense that they can not be trusted anymore, since they usually contain an arbitrary number of errors that are rooted in the first anomalous behavior exposed. Even if no additional errors are discovered, these *tainted* execution traces are not worth checking, since they would most likely change or even disappear as a result of fixing the current program error. So, we would better think of a kind of 'best fit' notion that allows us to prioritize the criterion (or criteria a) that will most likely lead to fixing a given error with less effort.

Finally, it is worth noting that the proposed framework can be easily adapted to check abstract computations such as trace slices in both an asynchronous way (i.e., by providing an input trace slice) or a synchronous way (i.e., by tuning the *expand* algorithm of Figure 4.5 and forwardly generating sliced execution steps). However, if an error is hidden within the sliced part of the computation, the analysis may not be able to reach it. Still, this approach can be useful to swiftly analyze more general properties of a program, since the state space to be cheked can be greatly reduced.

Although from a practical point of view this thesis specifically endows the Maude language with assertion checking capabilities, the proposed theoretical framework is general enough to serve other languages with minor adjustments. Even if functional assertions might seem at first sight strongly tied to Maude's equational reduction traces, they can be naturally used to check any confluent rule-based system or even non-confluent systems by exploring all possible derivations up to a certain depth to ensure termination. Adapting this framework to other programming languages is hence possible, although it technically depends on the characteristics of the target language.

9.3 Future Work

Assertion-driven computations can be easily accomplished by means of either assertion-guided execution strategies or assertion-based transformation techniques that essentially embed an assertional specification into the (potentially faulty) program so that any undesired behavior of the program is avoided. As a natural continuation of the work described in this thesis, a call-independent (static) transformation approach for assertion-driven computations in Maude is currently under investigation. It essentially consists of a transformation technique that fixes the program with respect to an assertional specification. Roughly speaking, the program fix is achieved by blending the assertions within the source code of the program. This is achieved by first transforming the assertions into suitable equations that are included into the program. Then, the newly introduced equations are used to strengthen the rules by imposing appropriate conditions that are checked by the new assertion-based equations. Among other advantages, the transformed programs can be executed without the need for external monitors that may burden their efficiency, and, moreover, their properties can be easily analyzed by existing tools that use the internal (rewriting and narrowing) execution strategies defined in the Maude system.

Bibliography

[Alba-Castro et al., 2010] Alba-Castro, M., Alpuente, M., and Escobar, S. (2010). Abstract Certification of Global Non-Interference in Rewriting Logic. In *Proceedings of the 8th International Symposium on Formal Methods for Components and Objects (FMCO 2009)*, volume 6286 of *Lecture Notes in Computer Science*, pages 105–124. Springer.

[Alpuente et al., 2008] Alpuente, M., Escobar, S., Meseguer, J., and Ojeda, P. (2008). A Modular Equational Generalization Algorithm. In *Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008)*, volume 5438 of *Lecture Notes in Computer Science*, pages 24–39. Springer.

[Alpuente et al., 2009a] Alpuente, M., Ballis, D., and Romero, D. (2009a). Specification and Verification of Web Applications in Rewriting Logic. In *Proceedings of the 16th International Symposium on Formal Methods (FM 2009)*, volume 5850 of *Lecture Notes in Computer Science*, pages 790–805. Springer.

[Alpuente et al., 2009b] Alpuente, M., Escobar, S., Meseguer, J., and Ojeda, P. (2009b). Order–Sorted Generalization. *Electronic Notes in Theoretical Computer Science*, 246:27–38.

[Alpuente et al., 2010a] Alpuente, M., Ballis, D., Baggi, M., and Falaschi, M. (2010a). A Fold/Unfold Transformation Framework for Rewrite Theories extended to CCT. In *Proceedings of the 19th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2010)*, pages 43–52. Association for Computing Machinery.

[Alpuente et al., 2010b] Alpuente, M., Ballis, D., Espert, J., and Romero, D. (2010b). Model-checking Web Applications with Web-TLR. In *Proceedings of the 8th International Symposium on Automated Technology for Verification and Analysis (ATVA 2010)*, volume 6252 of *Lecture Notes in Computer Science*, pages 341–346. Springer.

[Alpuente et al., 2011] Alpuente, M., Ballis, D., Espert, J., and Romero, D. (2011). Backward Trace Slicing for Rewriting Logic Theories. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE 2011)*, volume 6803 of *Lecture Notes in Computer Science*, pages 34–48. Springer.

[Alpuente et al., 2012] Alpuente, M., Ballis, D., Frechina, F., and Romero, D. (2012). Backward Trace Slicing for Conditional Rewrite Theories. In *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2012)*, volume 7180 of *Lecture Notes in Computer Science*, pages 62–76. Springer.

[Alpuente et al., 2013a] Alpuente, M., Ballis, D., Frechina, F., and Sapiña, J. (2013a). Slicing-Based Trace Analysis of Rewriting Logic Specifications with *i*Julienne. In *Proceedings of the 22nd European Symposium on Programming (ESOP 2013)*, volume 7792 of *Lecture Notes in Computer Science*, pages 121–124. Springer.

[Alpuente et al., 2013b] Alpuente, M., Ballis, D., Frechina, F., and Sapiña, J. (2013b). Parametric Exploration of Rewriting Logic Computations. In *Proceedings of the 5th International Symposium on Symbolic Computation in Software Science (SCSS 2013)*, volume 15 of *EasyChair Proceedings in Computing (EPiC)*, pages 4–18. EasyChair.

[Alpuente et al., 2014a] Alpuente, M., Ballis, D., Frechina, F., and Romero, D. (2014a). Using Conditional Trace Slicing for improving Maude Programs. *Science of Computer Programming*, 80, Part B:385 – 415.

[Alpuente et al., 2014b] Alpuente, M., Ballis, D., Frechina, F., and Sapiña, J. (2014b). Inspecting Rewriting Logic Computations (in a Parametric and Stepwise Way). In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi (SAS 2014)*, volume 8373 of *Lecture Notes in Computer Science*, pages 229–255. Springer.

[Alpuente et al., 2014c] Alpuente, M., Ballis, D., and Romero, D. (2014c). A Rewriting Logic Approach to the Formal Specification and Verification of Web Applications. *Science of Computer Programming*, 81:79–107.

[Alpuente et al., 2014d] Alpuente, M., Escobar, S., Espert, J., and Meseguer, J. (2014d). A Modular Order-Sorted Equational Generalization Algorithm. *Information and Computation*, 235:98–136.

[Alpuente et al., 2014e] Alpuente, M., Escobar, S., Espert, J., and Meseguer, J. (2014e). ACUOS: A System for Modular ACU Generalization with Subtyping and Inheritance. In *Proceedings of the 14th European Conference on Logics in Artificial Intelligence (JELIA 2014)*, volume 8761 of *Lecture Notes in Computer Science*, pages 573–581. Springer.

[Alpuente et al., 2015a] Alpuente, M., Ballis, D., Frechina, F., and Sapiña, J. (2015a). Exploring Conditional Rewriting Logic Computations. *Journal of Symbolic Computation*, 69:3–39.

[Alpuente et al., 2015b] Alpuente, M., Ballis, D., Frechina, F., and Sapiña, J. (2015b). Combining Runtime Checking and Slicing to improve Maude Error Diagnosis. In *Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer*, volume 9200 of *Lecture Notes in Computer Science*, pages 72–96. Springer.

[Alpuente et al., 2016a] Alpuente, M., Ballis, D., Frechina, F., and Sapiña, J. (2016a). Debugging Maude Programs via Runtime Assertion Checking and Trace Slicing. *Journal of Logical and Algebraic Methods in Programming*, 85:707–736.

[Alpuente et al., 2016b] Alpuente, M., Ballis, D., Frechina, F., and Sapiña, J. (2016b). Assertion-based Analysis via Slicing with ABETS. *Theory and Practice of Logic Programming*, 16(5–6):515–532.

[Alpuente et al., 2016c] Alpuente, M., Cuenca-Ortega, A., Escobar, S., and Meseguer, J. (2016c). Partial Evaluation of Order-sorted Equational Programs modulo Axioms. In *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*, volume 10184 of *Lecture Notes in Computer Science*, pages 3–20. Springer.

[Alpuente et al., 2017] Alpuente, M., Cuenca-Ortega, A., Escobar, S., and Sapiña, J. (2017). Inspecting Maude Variants with GLINTS. *Theory and Practice of Logic Programming*, 17(5–6):689–707.

[Antoy and Hanus, 2012] Antoy, S. and Hanus, M. (2012). Contracts and Specifications for Functional Logic Programming. In *Proceedings of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, volume 7149 of *Lecture Notes in Computer Science*, pages 33–47. Springer.

[Baader and Nipkow, 1998] Baader, F. and Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press.

[Bae and Meseguer, 2015] Bae, K. and Meseguer, J. (2015). Model Checking Linear Temporal Logic of Rewriting Formulas under Localized Fairness. *Science of Computer Programming*, 99:193–234.

[Barnett et al., 2004] Barnett, M., Rustan, K., Leino, M., and Schulte, W. (2004). The Spec# Programming System: An Overview. In *Proceedings of the 1st International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer.

[Barnett et al., 2011] Barnett, M., Fähndrich, M., Leino, K. R. M., Müller, P., Schulte, W., and Venter, H. (2011). Specification and verification: the Spec# experience. *Communications of the ACM*, 54(6):81–91.

[Barrett et al., 2011] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. (2011). CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer.

[Blume and McAllester, 2006] Blume, M. and McAllester, D. (2006). Sound and Complete Models of Contracts. *Journal of Functional Programming*, 16(4–5):375–414.

[Bruni and Meseguer, 2006] Bruni, R. and Meseguer, J. (2006). Semantic Foundations for Generalized Rewrite Theories. *Theoretical Computer Science*, 360(1–3):386–414.

[Burdy et al., 2003] Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J., Leavens, G. T., Leino, K. R. M., and Poll, E. (2003). An Overview of JML Tools and Applications. *Electronic Notes in Theoretical Computer Science*, 80:75–91.

[Burdy et al., 2005] Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M., and Poll, E. (2005). An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232.

[Chen and Roşu, 2009] Chen, F. and Roşu, G. (2009). Parametric Trace Slicing and Monitoring. In *Proceedings of the 15th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2009)*, volume 5505 of *Lecture Notes in Computer Science*, pages 246–261. Springer.

[Church and Rosser, 1936] Church, A. and Rosser, J. B. (1936). A Relational Model of Data for Large Shared Data Banks. *Transactions of the American Mathematical Society*, 39(3):472–482.

[Clarke and Rosenblum, 2006] Clarke, L. and Rosenblum, D. (2006). A Historical Perspective on Runtime Assertion Checking in Software Development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37.

[Clavel et al., 1996] Clavel, M., Eker, S., Lincoln, P., and Meseguer, J. (1996). Principles of Maude. In *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications (WRLA 1996)*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89. Elsevier Science.

[Clavel et al., 1999] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. F. (1999). The Maude System. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA 1999)*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243. Springer.

[Clavel et al., 2003] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2003). The Maude 2.0 System. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer.

[Clavel et al., 2007] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2007). *All About Maude: A High-Performance Logical Framework*. Springer.

[Clavel et al., 2015] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., and Talcott, C. (2015). Two Decades of Maude. In *Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer*, volume 9200 of *Lecture Notes in Computer Science*, pages 232–254. Springer.

[Clavel et al., 2016] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2016). Maude Manual (Version 2.7.1). Technical report, SRI International Computer Science Laboratory. Available at: `http://maude.cs.uiuc.edu/maude2-manual/`.

[Dershowitz, 1987] Dershowitz, N. (1987). Termination of Rewriting. *Journal of Symbolic Computation*, 3(1/2):69–116.

[Din et al., 2014] Din, C. C., Owe, O., and Bubel, R. (2014). Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014)*, pages 480–487. IEEE Computer Society Press.

[Durán, 1999] Durán, F. (1999). *A Reflective Module Algebra with Applications to the Maude Language*. Ph.D. thesis, University of Málaga.

[Durán and Meseguer, 2012] Durán, F. and Meseguer, J. (2012). On the Church-Rosser and Coherence Properties of Conditional Order-sorted Rewrite Theories. *The Journal of Logic and Algebraic Programming*, 81(7–8):816–850.

[Durán et al., 2016] Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2016). Built-in Variant Generation and Unification, and their Applications in Maude 2.7. In *Proceedings of the 8th International Joint Conference on Automated Reasoning (IJCAR 2016)*, volume 9706 of *Lecture Notes in Computer Science*, pages 183–192. Springer.

[Eker, 1995] Eker, S. (1995). Associative-Commutative Matching via Bipartite Graph Matching. *The Computer Journal*, 38(5):381–399.

[Eker, 2003] Eker, S. (2003). Associative-Commutative Rewriting on Large Terms. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 14–29. Springer.

[Frechina, 2014] Frechina, F. (2014). A Rewriting-based, Parameterized Exploration Scheme for the Dynamic Analysis of Complex Software Systems. Ph.D. thesis, Universitat Politècnica de València.

[Goguen et al., 1988] Goguen, J., Kirchner, C., Kirchner, H., Mégrelis, A., Meseguer, J., and Winkler, T. (1988). An Introduction to OBJ 3. In *Proceedings of the 1st International Workshop on Conditional Term Rewriting Systems (CTRS 1988)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer.

[Goguen and Meseguer, 1992] Goguen, J. A. and Meseguer, J. (1992). Order-sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theoretical Computer Science*, 105:217–273.

[Helmink et al., 1994] Helmink, L., Sellink, M. P. A., and Vaandrager, F. W. (1994). Proof-Checking a Data Link Protocol. In *Proceedings of the 4th International Workshop on Types for Proofs and Programs (TYPES 1993)*, volume 806 of *Lecture Notes in Computer Science*, pages 127–165. Springer.

[Hendrix et al., 2005] Hendrix, J., Clavel, M., and Meseguer, J. (2005). A Sufficient Completeness Reasoning Tool for Partial Specifications. In *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA 2005)*, volume 3467 of *Lecture Notes in Computer Science*, pages 165–174. Springer.

[Klop, 1992] Klop, J. (1992). Term Rewriting Systems. In Abramsky, S., Gabbay, D., and Maibaum, T., editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press.

[Korel and Laski, 1988] Korel, B. and Laski, J. (1988). Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163.

[Korel and Rilling, 1997] Korel, B. and Rilling, J. (1997). Application of Dynamic Slicing in Program Debugging. In *Proceedings of the 3rd International Workshop on Automated Debugging (AADEBUG 1997)*, pages 43–58. Linköping University Electronic Press.

[Lassez et al., 1988] Lassez, J. L., Maher, M. J., and Marriott, K. (1988). Unification Revisited. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, California.

[Leavens and Cheon, 2005] Leavens, G. T. and Cheon, Y. (2005). Design by Contract with JML. Available at: `http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf`.

[Leucker and Schallhart, 2009] Leucker, M. and Schallhart, C. (2009). A Brief Account of Runtime Verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303.

[Leucker, 2012] Leucker, M. (2012). Teaching Runtime Verification. In *Proceedings of the 2nd International Conference on Runtime Verification (RV 2011)*, volume 7186 of *Lecture Notes in Computer Science*, pages 34–48. Springer.

[Logozzo and Ball, 2012] Logozzo, F. and Ball, T. (2012). Modular and Verified Automatic Program Repair. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2012)*, pages 133–146. Association for Computing Machinery.

[Martelli and Montanari, 1982] Martelli, A. and Montanari, U. (1982). An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282.

[Martí-Oliet et al., 2012] Martí-Oliet, N., Palomino, M., and Verdejo, A. (2012). Rewriting Logic Bibliography by Topic: 1990–2011. *The Journal of Logic and Algebraic Programming*, 81(7–8):782–815.

[Mau-Dev, 2016] Mau-Dev (2016). The Mau-Dev Website. Available at: `http://safe-tools.dsic.upv.es/maudev`.

[Maude-Tools, 2010] Maude-Tools (2010). Maude Tools Website. Available at: `http://maude.cs.illinois.edu/w/index.php?title=Maude_Tools`.

[Mera et al., 2009] Mera, E., López-García, P., and Hermenegildo, M. V. (2009). Integrating software testing and run-time checking in an assertion verification framework. In *Proceedings of the 25th International Conference on Logic Programming (ICLP 2009)*, volume 5649 of *Lecture Notes in Computer Science*, pages 281–295. Springer.

[Meseguer, 1992] Meseguer, J. (1992). Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155.

[Meseguer, 1998] Meseguer, J. (1998). Membership Algebra as a Logical Framework for Equational Specification. In *Proceedings of the 12th International Workshop on Algebraic Development Techniques (WADT 1997)*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer.

[Meseguer, 2012] Meseguer, J. (2012). Twenty Years of Rewriting Logic. *The Journal of Logic and Algebraic Programming*, 81(7-8):721–781.

[Meyer, 1997] Meyer, B. (1997). *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall.

[Ohlebusch, 2002] Ohlebusch, E. (2002). *Advanced Topics in Term Rewriting*. Springer.

[Palamidessi, 1990] Palamidessi, C. (1990). Algebraic Properties of Idempotent Substitutions. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP 1990)*, volume 443 of *Lecture Notes in Computer Science*, pages 386–399. Springer.

[Plotkin, 1970] Plotkin, G. D. (1970). A Note on Inductive Generalization. *Machine Intelligence*, 5:153–163.

[Quesada, 1997] Quesada, J. F. (1997). *The SCP parsing algorithm based on syntactic constraint propagation*. Ph.D. thesis, University of Seville.

[Riesco et al., 2012] Riesco, A., Verdejo, A., Martí-Oliet, N., and Caballero, R. (2012). Declarative Debugging of Rewriting Logic Specifications. *The Journal of Logic and Algebraic Programming*, 81(7–8):851–897.

[Rocha et al., 2014] Rocha, C., Meseguer, J., and Muñoz, C. (2014). Rewriting Modulo SMT and Open System Analysis. In *Proceedings of the 10th International Workshop on Rewriting Logic and its Applications (WRLA 2014)*, volume 8663 of *Lecture Notes in Computer Science*, pages 247–262. Springer.

[Roşu, 2015] Roşu, G. (2015). From Rewriting Logic, to Programming Language Semantics, to Program Verification. In *Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer*, volume 9200 of *Lecture Notes in Computer Science*, pages 598–616. Springer.

[Roldán et al., 2009] Roldán, M., Durán, F., and Vallecillo, A. (2009). Invariant-driven Specifications in Maude. *Science of Computer Programming*, 74(10):812–835.

[Roldán and Durán, 2011] Roldán, M. and Durán, F. (2011). Dynamic Validation of OCL Constraints with mOdCL. *Electronic Communications of the EASST*, 44.

[Sapiña, 2013] Sapiña, J. (2013). Slicing Slices, an Incremental Backward Trace Slicing Methodology for RWL Computations. M.Sc. thesis, Universitat Politècnica de València.

[Shapiro, 1982] Shapiro, E. Y. (1982). Algorithmic Program Diagnosis. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1982)*, pages 299–308. Association for Computing Machinery.

[TeReSe, 2003] TeReSe (2003). *Term Rewriting Systems*. Cambridge University Press.

[Turing, 1937] Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265.

[Viry, 2002] Viry, P. (2002). Equational Rules for Rewriting Logic. *Theoretical Computer Science*, 285(2):487–517.

[Weiser, 1981] Weiser, M. (1981). Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE 1981)*, pages 439–449. IEEE Computer Society Press.

[Xu, 2012] Xu, D. N. (2012). Hybrid Contract Checking via Symbolic Simplification. In *Proceedings of the 21st ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2012)*, pages 107–116. Association for Computing Machinery.

Appendix A

A Stock Exchange System

```
fmod STOCK-EXCHANGE-SORTS is
  pr INT + QID .

  sorts Stock Trader Order Id Market PremiumTrader
        StockID TraderID OrderID  Set .
  subsorts Qid < StockID TraderID OrderID < Id .
  subsorts PremiumTrader < Trader .
endfm

view Stock from TRIV to STOCK-EXCHANGE-SORTS is
  sort Elt to Stock .
endv

view Trader from TRIV to STOCK-EXCHANGE-SORTS is
  sort Elt to Trader .
endv

view Order from TRIV to STOCK-EXCHANGE-SORTS is
  sort Elt to Order .
endv

view StockID from TRIV to STOCK-EXCHANGE-SORTS is
  sort Elt to StockID .
endv

mod RANDOMIZER is
  pr RANDOM .

  var X : Nat .
  op rndDelta : Nat -> Nat .
  eq [rnd-delta] : rndDelta(X) = random(X) rem 10 .
  op reSeed : Nat -> Nat .
  eq [re-seed] : reSeed(X) = X + 3 .
endm

mod STOCK-EXCHANGE is
  pr STOCK-EXCHANGE-SORTS + RANDOMIZER .
  pr SET{Stock} + SET{Trader} + SET{Order} + SET{StockID} .

  --- Round StockSet TraderSet OrderSet
  op _:_|_|_ : Nat Set{Stock} Set{Trader} Set{Order} ->
             Market [ctor] .

  --- TraderID capital
  op tr : TraderID Int -> Trader [ctor] .

  --- StockID price
  op st : StockID Int -> Stock [ctor] .

  --- OrderID TraderID StockID
  --- limit profit-target stop-loss active
  op ord : OrderID TraderID StockID Int Int Int Bool ->
         Order [ctor] .

  --- Round Seed StockSet
  op updP : Nat Nat Set{Stock} -> Set{Stock} .

  ops open closed : -> Bool .

  vars TID SID OID : Id .
  var SS : Set{Stock} .
  var TS : Set{Trader} .
  var OS : Set{Order} .
  var R S : Nat .
  vars P C L PT SL : Int .

  eq [updP] : updP(R,S,(st(SID,P),SS)) =
        if (rndDelta(R * S) rem 2) == 0
        then st(SID,S + rndDelta(R * S)),updP(R,S + 1,SS)
        else st(SID,S + (- rndDelta(R * S))),updP(R,S + 1,SS)
        fi .
  eq [updP-owise] : updP(R,S,empty) = empty [owise] .

  cmb [premT] : tr(TID,C) : PremiumTrader if TID
                in PreferredTraders .

  op PreferredTraders : -> Set{StockID} .
  eq [prefT] : PreferredTraders = 'T2 .

  rl [next-rnd] : R : SS | TS | OS =>
      R + 1 : updP(R + 1,reSeed(R + 1),SS) | TS | OS .

  crl [open-ord] :
      R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,SID,
      L,PT,SL,closed),OS)
      =>
      R : (st(SID,P),SS) | (tr(TID,C - P),TS) | (ord(OID,TID,
      SID,L,PT,SL,open),OS)
      if P <= L .

  crl [close-ord-SL] :
      R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,SID,
      L,PT,SL,open),OS)
      =>
      R : (st(SID,P),SS) | (tr(TID,C + L + (- SL)),TS) | OS
      if P <= L - SL .

  crl [close-ord-PT] :
      R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,SID,
      L,PT,SL,open),OS)
      =>
      R : (st(SID,P),SS) | (tr(TID,C + L + PT),TS) | OS
      if P >= L + PT .
endm
```

Appendix B

An Object-oriented Car Rental Store

```
(omod RENT-A-CAR-ONLINE-STORE is
  pr CONVERSION .
  pr QID .

  subsort Qid < Oid .

  class Register | rentals : Nat ,
                   date : Nat .

  class Customer | credit : Int,
                   suspended : Bool .

  class Car | available : Bool,
              rate : Nat .

  class Rental | deposit : Nat,
                 dueDate : Nat,
                 pickUpDate : Nat,
                 customer : Oid,
                 car : Oid .

  class PreferredCustomer .
  class EconomyCar .
  class MidSizeCar .
  class FullSizeCar .

  subclass PreferredCustomer < Customer .
  subclasses EconomyCar MidSizeCar FullSizeCar < Car .

  var B : Bool .
  vars U C R RG : Oid .
  vars CREDIT AMNT : Int .
vars TODAY PDATE DDATE RATE DPST RNTLS : Nat .

rl [new-day] : < RG : Register | date : TODAY >
            => < RG : Register | date : TODAY + 1 > .

crl [3-day-rental] :
  < U : Customer | credit : CREDIT, suspended : false >
  < C : Car | available : true, rate : RATE >
  < RG : Register | rentals : RNTLS, date : TODAY >
  =>
  < U : Customer | credit : CREDIT - AMNT >
  < C : Car | available : false >
  < RG : Register | rentals : RNTLS + 1 >
  < qid("R" + string(RNTLS,10)) : Rental | pickUpDate :
    TODAY, dueDate : TODAY + 3, car : C, deposit : AMNT,
    customer : U, rate : RATE >
  if AMNT := 3 * RATE .

crl [on-date-return] :
  < U : Customer | credit : CREDIT >
  < C : Car | rate : RATE >
  < R : Rental | customer : U, car : C, pickUpDate : PDATE,
    dueDate : DDATE, deposit : DPST >
  < RG : Register | date : TODAY >
  =>
  < U : Customer | credit : (CREDIT + DPST) - AMNT >
  < C : Car | available : true >
  < RG : Register | >
  if (TODAY <= DDATE) / AMNT := RATE * (TODAY - PDATE) .

crl [late-return] :
  < U : Customer | credit : CREDIT >
  < C : Car | rate : RATE >
  < R : Rental | customer : U, car : C, pickUpDate : PDATE,
    dueDate : DDATE, deposit : DPST >
  < RG : Register | date : TODAY >
  =>
  updateSuspension(< U : Customer | credit : (CREDIT - AMNT)
  + DPST >) < C : Car | available : true > < RG : Register
  | >
  if DDATE < TODAY / AMNT := RATE * (DDATE - PDATE) + (120
    * RATE * (TODAY - DDATE)) quo 100 .

  op updateSuspension : Object -> Object .
    ceq [suspend] : updateSuspension(
       < U : Customer | credit : CREDIT , suspended : false >)
       =
       < U : Customer | credit : CREDIT , suspended : true >
       if (CREDIT < 0) .

     eq [maintainSuspension] : updateSuspension(
       < U : Customer | suspended : B >)
       =
       < U : Customer | suspended : B > [owise] .
endom)
```

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIIC

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN