SYMBOLIC REACHABILITY ANALYSIS FOR REWRITE THEORIES

BY

HERNÁN CAMILO ROCHA NIÑO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor José Meseguer, Chair and Director of Research
Professor Kokichi Futatsugi, JAIST School of Information Science
Dr. César Muñoz, NASA Langley Research Center
Associate Professor Grigore Roşu
Associate Professor Mahesh Viswanathan

ABSTRACT

This dissertation presents a significant step forward in automatic and semi-automatic reasoning for *reachability properties* of rewriting logic specifications, a major research goal in the current state of the art. In particular, this work develops *deductive techniques* for reasoning *symbolically* about specifications with initial model semantics, including: (i) new constructor-based notions for reachability analysis, (ii) a proof system for the task of proving safety properties, and (iii) a novel method for symbolic reachability analysis of rewrite theories with constrained built-ins. These three new techniques are not just theoretical developments: each of them has been implemented in freely available tools for the automated reasoning presented in this thesis and are validated through case studies. These case studies include: (i) a reliable communication protocol, (ii) a secure-by-design browser system, and (iii) a NASA language for robotic machines. One main characteristic of the methods developed in this dissertation is that they are suitable for wide classes of rewrite theories and are highly generic, so that they can be used over many different instance languages and application domains.

To Laura, for making it all worthwhile.

ACKNOWLEDGMENTS

I would not have finished this dissertation without the advice, support, friendship, and encouragement of many people.

It is an honor to record my personal debt to Professor José Meseguer, whose sincere care and guidance made this work a reality. As an adviser, he has shown me the value of deep thinking. His research maturity, knowledge of the subject, and enthusiasm have been a source of inspiration for all the work surveyed in this manuscript. I am also grateful for the resolute financial support that he has provided me through research assistantships and conference travel. But above all, I wish to express my utmost admiration for his research passion, sense of responsibility, and personal values, especially in a society in which the ruthlessness of competition and greed has turned such virtues into unaffordable luxuries.

I am deeply indebted to César Muñoz for being an unconditional friend and a coauthor in many papers. He invited me to an exciting research project that is a major part of this dissertation. He arranged and funded many fruitful visits in Hampton at the National Institute of Aerospace where I wrote papers and Maude code, and met many interesting researchers. During my recurrent visits, I thoroughly enjoyed from his company and that of his wife Magda and two daughters, Laura Sofia and Isabela.

I thank the other members of my dissertation committee; they have all eagerly provided me with advice on both my research and career. I have learned from Kokichi Futatsugi the elegance of algebraic proofs; he also invited me to visit the Japan Advanced Institute of Science and Technology in Kanazawa for a week, where I gave the first talk about my approach for proving invariants. Grigore Rosu has taught me from the very first semester the practical importance of the algebraic method when formally studying

the semantics of programming languages. Mahesh Viswanathan mentored me on algorithmic verification methods, logic, and finite model theory.

I also thank my main research collaborators and other researchers who influenced my work. Francisco Durán for his support and friendship, for encouraging me to improve the state of the art in the Maude formal environment, for being a coauthor, and for inviting me to visit Málaga for three weeks. I must also thank my other coauthors José M. Álvarez, Héctor Cadavid, Gilles Dowek, Raúl Gutiérrez, and Radu Siminiceanu. I wish to take the opportunity to thank Jaime Bohórquez for inspiring me, through his teaching, to become a formal methods student in the first place. Many thanks are due to Santiago Escobar and Peter Csaba Ölveczky for sharing a friendship and many research ideas in years of visits and conversation.

Many thanks are due to Liza Mallozi Tapiero, manager of advising services at LASPAU, for her kindness and distant company during this journey. I am also grateful for the great staff at the University of Illinois and at the Escuela Colombiana de Ingeniería. Donna Coleman who has handled many different travel arrangements and administrative tasks with impeccable efficiency, and Mary Beth Kelly who has been extraordinarily helpful in meeting the requirements for finishing the degree. Patricia Castañeda, Patricia Salazar, and colleagues for their support from Colombia through the years.

I thank my office mates, fellow students, friends, and soccer buddies over the years for great conversations and memories. I especially thank Ralf Sasse whose friendship, support, and yearly taxation advise made my life easier. In also thank Musab AlTurki, Kyungmin Bae, Raúl Gutiérrez, Joe Hendrix, Mike Katelman, Andrew Cholewa, Si Liu, Stephen Skeirik, Fan Yang, Pavithra Prabhakar, Sruthi Bandhakavi, Rajesh Karmani, Traian Serbanuta, Andrei Stefanescu, Artur Boronat, Tanya Crenshaw, Azadeh Farzan, Bardia Sadri, Beatriz Alarcón, Sonia Santiago, Narciso Martí-Oliet, Hebert Herencia, Timo Latvala, Emerson Escobar, Alexander Agudelo, Beatriz Guerrero, Vasilis Paschalidis, Maria Masouraki, Andrés Ortiz, Meredith Hill, Álvaro Rojas, Victoria Flórez, Andrés Salas, Beatriz Mira, Manuel Baum, Jorge Garzón, Santiago Gutiérrez, Miles Johnson, Kyle Mathewson, Juan Medina, Camilo Quijano, Daniel Ribero, Sean Sivapalan, and Philip Slater. I apologize for any of the inevitable omissions in the above list.

Finally, I thank my parents Gilberto and Elisa, and siblings Oscar, Carolina, and Diana for their support, advice, and consistent encouragement. Most of all, I thank my wife Laura for her friendship, love, support, and patience. She has been there, through thick and through thin. For all those times, this thesis is dedicated to her.

TABLE OF CONTENTS

Page

CHAPTER 1

INTRODUCTION

Although this work expands many different areas, all of it relates to verifying the model theoretic satisfaction relation

$$\mathcal{T}_{\mathcal{R}} \models \varphi,$$

where $\mathcal{T}_{\mathcal{R}}$ is a *transition system* of some sort and φ is a *reachability property*. The transition system can describe the behavior of a communication protocol, the interactive session of a user with a web browser, or the simulation of a robotic machine. Then, the reachability property may refer to the fact that the communication protocol achieves reliable communication across a lossy channel and never deadlocks, or that the security policy in the web browser (protecting the user from malicious attacks) cannot be compromised even under the presence of such an attack, or that the robotic machine does not 'freeze' during a mission due to an unforeseen change of temperature in the environment. Actually, some of these scenarios are part of the case studies contained in this dissertation. But before entering into particular examples, let us look at $\mathcal{T}_{\mathcal{R}}$ and φ from a formal and general perspective.

Computer systems have become more powerful, and many different applications and services have grown to depend on them. This includes essential safety-critical systems such as communication networks and cyber-physical systems. In the end, these systems depend on the correct operation of computer hardware and the software controlling it. However, the software has become extraordinarily complex in order to deal with the diverse requirements of these different applications. Handling this development complexity while ensuring that the system satisfies all of its property requirements has become one of the greatest challenges in software development.

Fundamental to any system development and validation is a clear and

precise *semantics* that can be given to the meaning of both programs and properties. One prominent logical and semantic framework to doing this is *rewriting logic* [70], where a specification $\mathcal{R}$ has both a deduction-based operational semantics and an *initial model semantics* $\mathcal{T}_{\mathcal{R}}$. In this approach, programs are specified *axiomatically* by means of rewrite rules $l \to r$, and a state of a program is represented by a term t. Programs are *executed* by replacing instances of l appearing in t with the corresponding instance r, for each rule $l \to r$, meaning that t transitions to another state. Operational properties φ of a program $\mathcal{R}$ can then be expressed through modal logics such as various temporal logics, or in standard mathematical logic such as first-order logic.

A major research goal in the current state of the art and advanced in this thesis is to develop deductive techniques for reasoning symbolically about specifications with initial model semantics:

> *Deductive and symbolic verification methods for rewrite theories*, including narrowing-based methods, their combination with SMT solving, deductive temporal verification, and inductive proof methods ... New proof techniques, new algorithms, and new tool implementations are needed to make all this happen. The great advantage of developing them for suitable classes of rewrite theories is that they will be highly generic, so that they can be amortized over many different instance languages and application domains.
>
> José Meseguer
> *On some future research directions*
> 20 Years of Rewriting Logic [72].

Another important motivation for the deductive approach is that algorithmic methods such as model checking, although very-widely used, are not sufficient for all verification purposes. This is clear from the fact that satisfaction of properties is in general undecidable, from the infinite-state nature of many systems and, even when a system is finite-state for each initial state, from the fact that in general there may be an infinite number of initial states.

However, as the applications of rewriting logic have grown, it has been quite useful to add advanced features to the specification language such as reasoning modulo fundamental structural properties such as associativity and commutativity. Of course, these extra features can allow complex systems to be specified in a significantly simpler and more elegant way, but they poses a major challenge for *reasoning* about specifications. Both automated reasoning techniques and tools have to be built, or be extended, to

handle these more expressive features. A major point of this dissertation is to answer this challenge.

In particular, this work presents a significant step forward in automatic and semi-automatic reasoning for *reachability properties* of rewriting logic specifications as follows:

1. New constructor-based notions for reachability analysis are developed in Chapter 3.

2. A relatively complete proof system for the task of proving safety properties of rewrite theories is presented in Chapter 4.

3. A novel method for symbolic reachability analysis of rewrite theories with constrained built-ins is introduced in Chapter 7.

These three new techniques are not just theoretical developments: each of them has been implemented in freely available tools for the automated reasoning presented in this thesis and are validated through case studies. Specifically, this thesis presents the following case studies:

1. A reliable communication protocol in Chapter 5.

2. A secure-by-design browser system in Chapter 6.

3. A NASA language for robotic machines in Chapter 9.

Diagram 1.1 depicts the different topics in this dissertation and their logical connections. The diagram breaks the topics covered into theoretical results, tools, and case studies. The diagram also suggests a reading order and a conceptual division among the three main techniques for deductive and symbolic reasoning about reachability properties developed in this dissertation:

1. Constructor-based reachability analysis and deadlock freedom.

2. A deductive approach for proving safety properties.

3. Symbolic reachability analysis for theories with constrained built-ins.

1.1 Summary of Chapters and Contributions

This dissertation contributes to several ongoing research efforts within the areas of formal methods, algebraic specifications, deductive analysis, theorem proving, and symbolic reachability analysis.

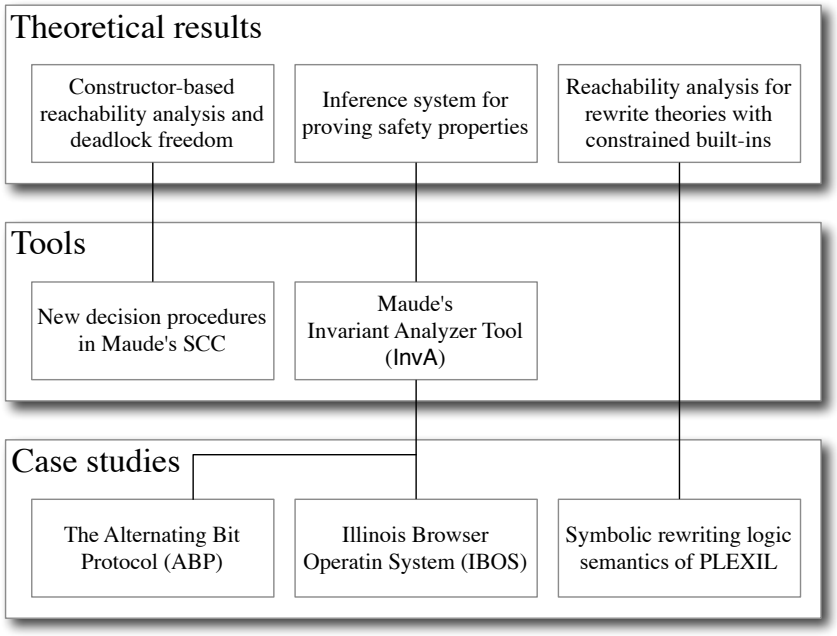

Figure 1.1: Relationship between contributions.

Chapter 3. A new notion of constructor for rewrite theories that generalizes the usual notion of constructor for equational specifications in the algebraic method. This notion turns out to be intimately related with the notion of deadlock freedom of a transition state system. It also makes possible *constructor-based inductive techniques for reachability analysis*. Decision procedures in the form of Propositional Tree Automata are exhibited for checking deadlock freedom and other notions, all implemented as extension of the Maude Sufficient Completeness Checker. The method of constructor-based inductive proof of reachability properties is summarized and illustrated with examples. This chapter is based on joint work with J. Meseguer [87, 86].

Chapter 4. A methodology and a proof system for proving safety properties of rewrite theories. The inference system is specialized to ground stability and ground invariance, and has rules for the application of strengthening techniques. The inference system has been implemented in the Maude Invariant Analyzer Tool (InvA), which offers great degree of automation for discharging proof obligations. This chapter is based on joint work with J. Meseguer [89, 88].

Chapter 5. The Alternating Bit Protocol (ABP), a well-established benchmark in the area of mechanical reasoning for concurrent systems, is

mechanically proved correct with help of the methods of Chapter 4 and the InvA implementation. This chapter is based on joint work with J. Meseguer

Chapter 6. The Illinois Browser Operating System (IBOS), a state-of-the-art browsing system designed with the idea of security in mind, is analyzed for safety properties with help of the InvA tool. It is proved automatically that the web browser satisfies some security invariants of interest by discharging thousands of proof obligations. The case study served also as a stress test for InvA, given the large size of this specification. Some areas that have room for improvement in the InvA are identified and left for future work. This chapter is based on joint work with J. Meseguer.

Chapter 7. A sound and complete approach for analyzing rewrite theories with constrained built-in terms is presented. The main feature of this symbolic method is that, with the help of SMT solving techniques, it can be based on matching instead than on unification. This approach is specially suitable for symbolically analyzing reachability properties of rewrite theories modulo decision procedures such as those supported by SMT solving. This chapter is based on joint work with C. Muñoz.

Chapter 8. A rewriting logic semantics of NASA's Plan Execution Interchange Language (PLEXIL), a benchmark for the official interpreter of the language. PLEXIL was designed by NASA to meet the requirements of flexible, efficient, and reliable plan execution in space mission operations. This chapter is based on joint work with H. Cadavid, G. Dowek, C. Muñoz, and R. Siminiceanu [31, 91, 84, 90].

Chapter 9. An implementation of the symbolic techniques introduced in Chapter 7 for the symbolic analysis of PLEXIL plans, thus complementing the more limited kinds of reachability and model checking analysis possible using the semantics for ground plans in Chapter 8. The non-determinism for the PLEXIL language is modeled by symbolic variables that are left unspecified or are partially specified with Boolean constraints. This chapter is based on joint work with C. Muñoz.

Chapter 10. Concluding remarks and opened research directions.

CHAPTER 2

PRELIMINARIES

This thesis uses standard notation and terminology about terms, term algebras, and order-sorted equational theories as employed, for example, by [5] and [46].

2.1 Order-Sorted Equational and Rewrite Theories

An *order-sorted signature* Σ is a tuple $\Sigma = (S, \leq, F)$ with a finite poset of sorts $(S, \leq)$ and set of function symbols F. The binary relation $\equiv_\leq$ denotes the equivalence relation generated by $\leq$ on S and its point-wise extension to strings in S^*. The function symbols in F can be subsort-overloaded and satisfy the condition that, for $(w, s), (w', s') \in S^* \times S$, if $f \in F_{w,s} \cap F_{w',s'}$, then $w \equiv_\leq w'$ implies $s \equiv_\leq s'$. A *top sort* in Σ is a sort $s \in S$ such that if $s' \in S$ and $s \equiv_\leq s'$, then $s' \leq s$. For any sort $s \in S$, the expression $[s]$ denotes the connected component of s, that is, $[s] = [s]_{\equiv_\leq}$, called the kind of the connected component. A signature Σ can be *kind-completed* by adding to it: (i) a new top sort $[s]$ above all sorts $s \in S$, and (ii) a new subsort-overloaded $f : [s_1] \cdots [s_n] \longrightarrow [s]$ for each $f : s_1 \cdots s_n \longrightarrow s$ in Σ.

The collection of *variables* X is an S-indexed family $X = \{X_s\}_{s \in S}$ of disjoint variable sets with each X_s countably infinite. The *set of terms of sort* s is denoted $T_\Sigma(X)_s$ and the *set of ground terms of sort* s is denoted $T_{\Sigma,s}$. The expressions $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$ denote the corresponding order-sorted Σ-term algebras. It is assumed that all order-sorted signatures are *preregular* [46], i.e., each Σ-term has a *least sort* $ls(t) \in S$ such that $t \in T_\Sigma(X)_{ls(t)}$. A term is called *linear* if no variable occurs in it twice. The *set of variables* of a term t is written $vars(t)$ and is extended to sets of terms in the natural way.

A *position* in a term is denoted by strings of natural numbers, indicating the sequences of branches from the root to each subterm. The expression $pos(t)$ denotes the collection of *positions* of $t \in T_\Sigma(X)$. Given a position $\pi \in pos(t)$, the expressions t_π and $\pi(t)$ denote, respectively, the subterm of t occurring at position π, and the topmost operator in t_π. For ϵ, the *empty position*, t_ϵ denotes the whole term t. Given a set C of function symbols, $pos_C(t)$ denotes the set of positions of the subterms of t whose root symbol is in C, that is, $pos_C(t) = \{\pi \in pos(t) \,|\, \pi(t) \in C\}$. By definition, $pos_\Sigma(t) = pos_F(t)$, for all $t \in T_\Sigma(X)$.

A *substitution* is an S-indexed mapping $\theta : X \longrightarrow T_\Sigma(X)$ that maps variables of sort s to terms of sort s and is different from the identity for a finite subset of X. The identity substitution is denoted by id and the expression $\theta|_Y$ denotes the restriction of a substitution θ to a set of variables $Y \subseteq X$. The expression $ran(\theta)$ denotes the set of variables introduced by θ. Substitutions extend homomorphically to terms in the natural way. A substitution θ is called *ground* if and only if $ran(\theta) = \varnothing$. The application of a substitution θ to a term t is denoted by $t\theta$ and the composition of two substitutions θ_1 and θ_2 is denoted by $\theta_1\theta_2$. A *context* C is a λ-term of the form $C = \lambda x_1, \ldots, x_n.c$ with $c \in T_\Sigma(X)$ and $\{x_1, \ldots, x_n\} \subseteq vars(c)$. A context C can be viewed as a n-ary function $C(t_1, \ldots, t_n) = c\theta$, where $\theta(x_i) = t_i$ for $1 \leq i \leq n$ and $\theta(x) = id(x)$ otherwise. Given a sort $s \in S$, a context $C = \lambda x_1, \ldots, x_n.c$ is called an *s-context* if and only if $\{x_1, \ldots, x_n\} \subseteq X_s$ for $s \in S$.

2.1.1 Equational Theories and Initial Algebras

A Σ-*equation* is an unoriented pair $t = u$ with $t \in T_\Sigma(X)_{s_t}$, $u \in T_\Sigma(X)_{s_u}$, and $s_t \equiv_\leq s_u$. A *conditional* Σ-*equation* is a triple $t = u$ **if** γ, with $t = u$ a Σ-equation and γ a finite conjunction of Σ-equations; it is called *unconditional* if γ is the empty conjunction. An *equational theory* is a tuple (Σ, E), with Σ an order-sorted signature and E a finite collection of (possibly conditional) Σ-equations. Throughout this thesis, it is assumed that $T_{\Sigma,s} \neq \varnothing$ for each $s \in S$, because this affords a simpler deduction system. An equational theory $\mathcal{E} = (\Sigma, E)$ induces the congruence relation $=_\mathcal{E}$ on $T_\Sigma(X)$ defined for $t, u \in T_\Sigma(X)$ by $t =_\mathcal{E} u$ if and only if $\mathcal{E} \vdash t = u$ by the deduction rules for order-sorted equational logic in [71], if and only if, [71] $t = u$ is valid in all models of $\mathcal{E}$. The *$\mathcal{E}$-subsumption* ordering $\ll_\mathcal{E}$ is the binary relation on $T_\Sigma(X)$ defined for any $t, u \in T_\Sigma(X)$ by $t \ll_\mathcal{E} u$ if and only if there is a substitution $\theta : X \longrightarrow T_\Sigma(X)$ such that $t =_\mathcal{E} u\theta$. A set of equations E is

called *collapse-free* for a sort $s \in S$ if and only if for any $x \in X_s$ E does not contain any equations of the form either $t = x$ **if** γ or $x = t$ **if** γ.

The expressions $\mathcal{T}_{\mathcal{E}}(X)$ and $\mathcal{T}_{\mathcal{E}}$ (or similarly $\mathcal{T}_{\Sigma/E}(X)$ and $\mathcal{T}_{\Sigma/E}$) denote the quotient algebras induced by $=_{\mathcal{E}}$ on the term algebras $\mathcal{T}_{\Sigma}(X)$ and $\mathcal{T}_{\Sigma}$, respectively. The algebra $\mathcal{T}_{\Sigma/E}$ is called the *initial algebra* of (Σ, E). A theory inclusion $(\Sigma, E) \subseteq (\Sigma', E')$, with $\Sigma \subseteq \Sigma'$ and $E \subseteq E'$, is called *protecting* if the unique Σ-homomorphism $\mathcal{T}_{\Sigma/E} \longrightarrow \mathcal{T}_{\Sigma'/E'}|_{\Sigma}$ to the Σ-reduct of the initial algebra $\mathcal{T}_{\Sigma'/E'}$ is a Σ-isomorphism, written $\mathcal{T}_{\Sigma/E} \simeq \mathcal{T}_{\Sigma'/E'}|_{\Sigma}$. A set of equations E is called *regular* if and only if $vars(t) = vars(u)$ for any equation $t = u$ **if** $\gamma \in E$. For $t, u \in T_{\Sigma}(X)$, the expression $GU_{\mathcal{E}}(t = u)$ denotes the *set of ground $\mathcal{E}$-unifiers of* $t = u$, i.e., $GU_{\mathcal{E}}(t = u) = \{\sigma : X \longrightarrow T_{\Sigma} \mid t\sigma =_{\mathcal{E}} u\sigma\}$.

2.1.2 Rewrite Theories and Initial Reachability Models

A Σ-*sequent* is an oriented pair $t \rightarrow u$ with $t \in T_{\Sigma}(X)_{s_t}$, $u \in T_{\Sigma}(X)_{s_u}$, and $s_t \equiv_{\leq} s_u$. A *conditional* Σ-*rule* is a triple $t \rightarrow u$ **if** γ, with $t \rightarrow u$ Σ-sequent satisfying $vars(u) \subseteq vars(t)$ and γ a finite conjunction of Σ-equations; it is called *unconditional* if γ is the empty conjunction. A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E, R)$ with equational theory $\mathcal{E}_{\mathcal{R}} = (\Sigma, E)$ and a finite set of Σ-rules R. A *topmost rewrite theory* is a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, such that each rule $t \rightarrow u$ **if** $\gamma \in R$ is such that $l, r \in T_{\Sigma}(X)_s$ for some top sort $s = [s]$ in Σ, $l \notin X$, and no operator in Σ has s as argument sort. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ induces the rewrite relation $\rightarrow_{\mathcal{R}}$ on $T_{\Sigma}(X)$ defined for $t, u \in T_{\Sigma}(X)$ by $t \rightarrow_{\mathcal{R}} u$ if and only if a *one-step* rewrite proof of $\mathcal{R} \vdash t \rightarrow u$ can be obtained by the deduction rules for order-sorted rewrite theories in [18], if and only if, [18] $t \rightarrow u$ is valid in all models of $\mathcal{R}$. For $t, u \in T_{\Sigma}(X)$, $\mathcal{R} \vdash t = u$ if and only if $\mathcal{E}_{\mathcal{R}} \vdash t = u$.

The expression $\mathcal{T}_{\mathcal{R}} = (\mathcal{T}_{\Sigma/E}, \rightarrow_{\mathcal{R}}^{*})$ denotes the *initial reachability model* of $\mathcal{R} = (\Sigma, E, R)$ [18], with $\rightarrow_{\mathcal{R}}^{*}$ expressing the reflexive-transitive closure of $\rightarrow_{\mathcal{R}}$. A Σ-sequent $t \rightarrow u$ is an *inductive consequence* of $\mathcal{R}$, written $\mathcal{R} \Vdash t \rightarrow u$, if and only if $R \vdash t\sigma \rightarrow u\sigma$ for each ground substitution $\sigma : X \longrightarrow T_{\Sigma}$ if and only if $\mathcal{T}_{\mathcal{R}} \models t \rightarrow u$.

2.2 Admissible Modules in Maude

The *Maude* tool [23] is a high-performance implementation of rewriting logic. It supports order-sorted equational specification in *functional modules*, cor-

responding to equational theories $\mathcal{E}$, and order-sorted rewrite specification in *system modules*, corresponding to full rewrite theories $\mathcal{R}$. In functional modules other functional modules can be included, sorts and subsorts can be declared, and operator symbols can be defined, possibly with equational attributes (called *axioms*). Sorts, subsorts, and conditional equations define the computations that are possible. In system modules other functional and system modules can be included, and the rewrite rules define the system transitions that are possible.

2.2.1 Admissible Functional Modules

Reasonable executability requirements are needed to make a module *admissible* (see [23], Sections 4.6 and 6.3). It is assumed that the set of Σ-equations of an equational theory $\mathcal{E}$ can be decomposed into a disjoint union $E \cup B$, with B a collection of structural axioms (such as associativity, and/or commutativity, and/or identity) for which there exists a *matching algorithm modulo* B producing a finite number of B-matching solutions, or failing otherwise. It is also assumed that the equations E can be oriented into a set (of possibly conditional) sort-decreasing, operationally terminating, and confluent conditional rewrite rules $\overrightarrow{E}$ modulo B. The set $\overrightarrow{E}$ is *sort decreasing* modulo B if and only if for each $t \to u$ **if** $\gamma \in \overrightarrow{E}$ and substitution θ, $ls(t\theta) \geq ls(u\theta)$ if $(\Sigma, B, \overrightarrow{E}) \vdash \gamma\theta$. The set $\overrightarrow{E}$ is *operationally terminating* modulo B if and only if there is no infinite well-formed proof tree in $(\Sigma, B, \overrightarrow{E})$ [32]. The set $\overrightarrow{E}$ is *confluent* modulo B if and only if for all $t, t_1, t_2 \in T_\Sigma(X)$, if $t \to^*_{E/B} t_1$ and $t \to^*_{E/B} t_2$, then there exists $u \in T_\Sigma(X)$ such that $t_1 \to^*_{E/B} u$ and $t_2 \to^*_{E/B} u$. The term $t \downarrow_{E/B} \in T_\Sigma(X)$ denotes the *E-canonical form* of t modulo B (or *E/B-canonical form*) so that $t \to^*_{E/B} t \downarrow_{E/B}$ and $t \downarrow_{E/B}$ is $\to_{E/B}$*-irreducible*, i.e., it cannot be further reduced by $\to_{E/B}$. Under the above assumptions $t \downarrow_{E/B}$ is unique up to B-equality. Then, Maude can execute an admissible functional module by equational simplification modulo the axioms, where the equations in E are used as rules from left to right and Maude's built-in matching for the axioms B leads for each term t to its canonical form with a least sort. In particular, this yields an operational semantics defined by the algebra of canonical forms $\mathbf{Can}_{\Sigma/E \cup B}$, which under the above admissibility assumptions, is isomorphic to the initial algebra $\mathcal{T}_{\Sigma/E \cup B}$. Equational simplification modulo axioms is executed by the `reduce` command in Maude.

2.2.2 Admissible System Modules

In order to be admissible, a system module corresponding to a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ has to, in addition to its equational component being admissible, satisfy the *coherence* requirement [34]. Coherence precisely means that Maude can execute an admissible system module by adopting the strategy of first simplifying a term t to its E/B-canonical form and then applying a rule with R modulo B to achieve the effect of rewriting with R modulo $E \cup B$. This is exactly the mathematical semantics of $\mathcal{R}$. In particular, this yields an operational semantics defined by the canonical reachability model $\mathbf{Can}_{\mathcal{R}}$ isomorphic, under the above admissibility requirements, to the initial reachability model $\mathcal{T}_{\mathcal{R}}$. Rewrites in a system module are performed in Maude by the `rewrite` command. There is also a breadth-first `search` command, and a built-in linear temporal logic (LTL) model checker to verify safety and liveness properties.

2.3 Order-Sorted Equality Enrichments Modulo Axioms

The use of equality enrichments is pervasive throughout this thesis. This section summarizes the main properties of equality enrichments; see [49, 48] for more details about their properties and their effective implementation for a wide class of order-sorted equational theories with free constructors modulo axioms. Given an order-sorted signature $\Sigma = (S, \leq, F)$ and an order-sorted equational theory $\mathcal{E} = (\Sigma, E)$ with initial algebra $\mathcal{T}_{\mathcal{E}}$, an *equality enrichment* [73] of $\mathcal{E}$ is an equational theory $\mathcal{E}^{\sim}$ that extends $\mathcal{E}$ by defining a Boolean-valued equality function symbol '$\sim$' that coincides with '$=$' in $\mathcal{T}_{\mathcal{E}}$.

Definition 1. *An equational theory $\mathcal{E}^{\sim} = (\Sigma^{\sim}, E^{\sim})$ is called an* equality enrichment *of $\mathcal{E} = (\Sigma, E)$, with $\Sigma^{\sim} = (S^{\sim}, \leq^{\sim}, F^{\sim})$ and $\Sigma = (S, \leq, F)$, if and only if*

- *$\mathcal{E}^{\sim}$ is a protecting extension of $\mathcal{E}$;*
- *the poset of sorts of $\Sigma^{\sim}$ extends $(S, \leq)$ by adding a new sort Bool that belongs to a new connected component, with constants $\top$ and $\bot$ such that $\mathcal{T}_{\mathcal{E}^{\sim}, Bool} = \{[\top], [\bot]\}$, with $\top \neq_{E^{\sim}} \bot$; and*
- *for each connected component in $(S, \leq)$ there is a top sort $k \in S^{\sim}$ and a binary commutative operator $\_ \sim \_ : k\ k \longrightarrow Bool$ in $\Sigma^{\sim}$, such that*

the following equivalences hold for any ground terms $t, u \in T_{\Sigma,k}$*:*

$$\mathcal{E} \vdash t = u \quad \Longleftrightarrow \quad \mathcal{E}^{\sim} \vdash (t \sim u) = \top, \tag{2.1}$$

$$\mathcal{E} \nvdash t = u \quad \Longleftrightarrow \quad \mathcal{E}^{\sim} \vdash (t \sim u) = \bot. \tag{2.2}$$

An equality enrichment $\mathcal{E}^{\sim}$ of $\mathcal{E}$ is called Boolean *if and only if it contains all the function symbols and equations making the elements of $\mathcal{T}_{\mathcal{E}^{\sim},Bool}$ a two-element Boolean algebra.*

The equality predicate '$\sim$' in $\mathcal{E}^{\sim}$ is sound for inferring equalities and inequalities in the initial algebra $\mathcal{T}_{\mathcal{E}}$, even for terms with variables. The precise meaning of this claim is given by Proposition 1.

Proposition 1 (Equality Enrichment Properties)**.** *Let $\mathcal{E}^{\sim} = (\Sigma^{\sim}, E^{\sim})$ be an equality enrichment of $\mathcal{E} = (\Sigma, E)$. If $t, u \in T_{\Sigma}(X)$, then the following equivalences hold:*

$$\mathcal{T}_{\mathcal{E}} \models (\forall X)\ t = u \quad \Longleftrightarrow \quad \mathcal{T}_{\mathcal{E}^{\sim}} \models (\forall X)\ (t \sim u) = \top, \tag{2.3}$$

$$\mathcal{T}_{\mathcal{E}} \models (\exists X)\ \neg(t = u) \quad \Longleftrightarrow \quad \mathcal{T}_{\mathcal{E}^{\sim}} \models (\exists X)\ (t \sim u) = \bot, \tag{2.4}$$

$$\mathcal{T}_{\mathcal{E}} \models (\forall X)\ \neg(t = u) \quad \Longleftrightarrow \quad \mathcal{T}_{\mathcal{E}^{\sim}} \models (\forall X)\ (t \sim u) = \bot. \tag{2.5}$$

By using an equality enrichment $\mathcal{E}^{\sim}$ of $\mathcal{E}$, the problem of reasoning in $\mathcal{T}_{\mathcal{E}}$ about a universally quantified inequality $\neg(t = u)$ (abbreviated $t \neq u$) can be reduced to reasoning in $\mathcal{T}_{\mathcal{E}^{\sim}}$ about the universally quantified equality $(t \sim u) = \bot$. A considerably more general reduction, not just for inequalities but for *arbitrary quantifier-free first-order formulae*, can be obtained with Boolean equality enrichments.

Corollary 1. *Let $\mathcal{E}^{\sim} = (\Sigma^{\sim}, E^{\sim})$ be a Boolean equality enrichment of $\mathcal{E} = (\Sigma, E)$. Let $\varphi = \varphi(t_1 = u_1, \ldots, t_n = u_n)$ be a quantifier-free Boolean formula whose atoms are the Σ-equations $t_i = u_i$ with variables in X, for $1 \leq i \leq n$, and with Boolean connectives in $\{\neg, \vee, \wedge\}$. Then, the following equivalence holds:*

$$\mathcal{T}_{\mathcal{E}} \models (\forall X)\varphi \quad \Longleftrightarrow \quad \mathcal{T}_{\mathcal{E}^{\sim}} \models (\forall X)\ \widehat{\varphi}(t_1 \sim u_1, \ldots, t_n \sim u_n) = \top, \tag{2.6}$$

where $\widehat{\varphi}(t_1 \sim u_1, \ldots, t_n \sim u_n)$ is the $\Sigma^{\sim}$-term of sort Bool obtained from φ by replacing each occurrence of the logical connectives $\neg$, $\vee$, and $\wedge$ by, respectively, the function symbols $\multimap$, $\sqcup$, and $\sqcap$ in $\mathcal{E}^{Bool}$, and each occurrence of an atom $t_i = u_i$ by the Bool term $t_i \sim u_i$, for $1 \leq i \leq n$.

In this thesis the Boolean theory $\mathcal{E}^{Bool}$ specified in [23, Subsection 9.1] is used. The theory $\mathcal{E}^{Bool}$ has free constructors modulo B^{Bool}, it is sort-decreasing, confluent, and operationally terminating modulo associativity-commutativity axioms, and hence provides a Boolean decision procedure. It has signature of free constructors $\Omega^{Bool} = \{\top, \bot\}$, set of defined symbols $\Sigma^{Bool} \setminus \Omega^{Bool} = \{ \lrcorner, \sqcap, \sqcup, \oplus, \supset \}$, and satisfies $\mathcal{T}_{\mathcal{E}^{Bool}} \models \top \neq \bot$. The choice of $\mathcal{E}^{Bool}$ is somewhat arbitrary: any equational theory implementing an equational Boolean decision procedure should suffice for the purpose here (for instance, see [85] for other equational Boolean decision procedures).

2.4 The Illinois Browser Operating System (IBOS)

The Illinois Browser Operating System (IBOS) [101] is a modern, security-conscious web browser designed at the University of Illinois which could be integrated into a secure operating system. The basic idea is to move from the monolithic approach and modularize the different processes of the browser. There is only one truly trusted process, the kernel. All other process such as web page instances, network processes, storage, etc., are not trusted. Security of all uncompromised components is desired, even when there are some compromised components in the mix. For that reason, all communication must go through the kernel, which will allow or disallow it based on its specific policies. See Chapter 6 for more details.

2.5 NASA's PLEXIL Language

The Plan Execution Interchange Language (PLEXIL) [39] is a language developed by NASA for representing plans for automation and a technology for executing these plans on real or simulated systems. PLEXIL was designed to meet the requirements of flexible, efficient and reliable plan execution in space mission operations. It is compact, semantically clear, and deterministic given the same sequence of events from the external world. At the same time, the language is quite expressive and can represent branches, loops, time- and event- driven activities, concurrent activities, sequences, and temporal constraints. The core syntax of the language is simple and uniform, making plan interpretation simple and efficient, while enabling the application of validation and testing techniques. See chapters 8 and 9 for more details.

2.6 The CETA Library

CETA [52] is a library for reasoning about Boolean combinations of equational tree languages. It supports emptiness testing of tree languages definable by a Boolean combination of regular tree automata over an equational theory containing operators that are associative and/or commutative and maybe have identity symbols. CETA is based on *propositional tree automata* (PTA) [52] and offers algorithms and data structures for representing tree automata, combining tree automata using Boolean operations, and testing emptiness. The decision procedures for checking deadlock freedom of rewrite theories in this thesis are based on PTA and have been implemented using the CETA library. See Chapter 3 for more details.

2.7 The Maude ITP

The Maude ITP [24, 52] is an experimental interactive tool for proving properties of the initial algebra $\mathcal{T}_{\mathcal{E}}$ of an order-sorted equational theory $\mathcal{E}$ written in Maude. The ITP has been written entirely in Maude and it is in fact an *executable* specification in Membership Equational Logic (MEL) [71], an equational super-logic of order-sorted equational logic, of the formal inference system that it implements. It supports different induction principles for terms including structural and coverset induction. Some equational inductive obligations in this thesis have been proved using the ITP tool. See Chapter 4 and Chapter 6 for more details.

CHAPTER 3

CONSTRUCTORS AND DEADLOCK FREEDOM

This chapter is concerned with the *sufficient completeness* and *deadlock freedom* of rewrite theories, with automatic proof methods for checking these properties, and with the closely related topic of *constructor-based* symbolic reachability analysis.

Sufficient completeness has been thoroughly studied for equational specifications, where function symbols are classified into *constructors* and *defined* symbols. But what should sufficient completeness mean for a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ with equations E and non-equational rules R describing concurrent transitions in a system? Since a rewrite theory comprises deduction with both equations E and rules R, this chapter argues that there are *two different notions of constructors* for $\mathcal{R}$ and therefore two different notions of sufficient completeness with quite different meanings:

- *Equational constructors*, or *E-constructors*, are specified by a subsignature $\Omega \subseteq \Sigma$, and then *E-sufficient completeness* is the usual requirement that for each sort s and each ground term $t \in T_\Sigma$ of that sort there is a ground term $u \in T_\Omega$ of sort s such that $(\Sigma, E) \vdash t = u$.

- *Rewrite constructors*, or *$\mathcal{R}$-constructors*, are specified by a subsignature $\Upsilon \subseteq \Sigma$, and then *$\mathcal{R}$-sufficient completeness* is the different requirement that for each sort s and each ground term $t \in T_\Sigma$ of that sort there is a ground term $v \in T_\Upsilon$ of sort s such that $\mathcal{R} \vdash t \to v$.

Intuitively, *E-sufficient completeness* has the traditional meaning in which function symbols in $\Sigma \setminus \Omega$ are *fully defined* by means of the equations E, so that any ground term can be proved equal by E to one where only operators in Ω are used. But how should $\mathcal{R}$-sufficient completeness be intuitively understood?

First of all, because of rewriting logic's equality rule (see [18]), whenever there is a proof of $(\Sigma, E) \vdash t = u$ there is also a (zero-step) proof of $\mathcal{R} \vdash t \rightarrow u$. That is, since the states of $\mathcal{R}$ are E-equivalence classes of terms $[t]_E$, there is already a representative term $u \in [t]_E$ with $u \in T_\Omega$, so that E-constructors are *trivially* $\mathcal{R}$-constructors. Therefore, for $\mathcal{R}$-constructors to have any teeth, a more restrictive subsignature $\Upsilon \subseteq \Omega$ is needed, so that each ground Σ-term of a given sort reaches *nontrivially* a ground Υ-term of the same sort. $\mathcal{R}$-sufficient completeness then provides an algebraic notion of *deadlock freedom*, that is, of *proper termination*. A concurrent system design often has an intended set P of *goal states* that any computation should ultimately reach. A system is then called *deadlock-free* outside P if and only if all *terminal* system states belong to P. Therefore, $\mathcal{R}$-sufficient completeness implies that $\mathcal{R}$ is *deadlock free outside* T_Υ.

It is well-known that E-constructors are essential for *inductive equational reasoning*, i.e., reasoning about the theorems satisfied by the initial algebra $\mathcal{T}_{\Sigma/E}$. This chapter argues that $\mathcal{R}$-constructors (and also E-constructors) play a similarly crucial role in reasoning about *inductive reachability properties* of the initial model $\mathcal{T}_\mathcal{R}$ of the rewrite theory $\mathcal{R}$, which intuitively models the states and concurrent computations of the system defined by $\mathcal{R}$.

This chapter also investigates *automatic sufficient completeness proof methods* based on equational tree automata under appropriate left-linearity assumptions, and it reports on their implementation in an extension of Maude's Sufficient Completeness Checker (SCC) [52]. The need for equational tree automata, as opposed to just standard tree automata, comes from the fact that the equations E in many rewrite theories $\mathcal{R} = (\Sigma, E, R)$ naturally decompose as a union $E = E_0 \cup B$, where B is a set of structural axioms such as associativity, and/or commutativity, and/or identity for some operators in Σ, and the equations E_0 are (ground) sort-decreasing, confluent, and operationally terminating modulo B.

One last contribution of this chapter is to generalize the notions of constructors, sufficient completeness, deadlock freedom, the equational tree automata methods of checking sufficient completeness and deadlock freedom, and the role of $\mathcal{R}$ constructors (and E-constructors) in reasoning about inductive reachability and joinability properties to the case of generalized rewrite theories of the form $\mathcal{R} = (\Sigma, E, R, \nu)$ (see [18]). The additional component ν maps each operator f or n arguments to a subset $\nu(f) \subseteq \{1, \ldots, n\}$ of its *frozen* argument positions, so that rewriting with R under such positions is forbidden. Note that a standard rewrite theory $\mathcal{R} = (\Sigma, E, R)$ can now be seen as the special case $\mathcal{R} = (\Sigma, E, R, \bot)$, where $\bot(f) = \varnothing$ for each

function symbol f. Such a frozenness map ν is very natural in various applications; therefore a more general theoretical treatment in this form is given. This generalization achieves, for the sufficient completeness of rules R with frozenness constraints ν, proof methods (and tool support in the extended version of the Maude SCC presented here), which are similar to those developed in [52, 54] at the equational level for algebraic specifications where the equations E are applied with a context-sensitive rewriting strategy map.

This chapter is organized as follows. Section 3.1 gathers preliminaries on the more general case of generalized rewrite theories assumed in this section. Section 3.2 gives the main definitions and theorems about sufficient completeness and deadlock freedom. A class of generalized rewrite theories with simpler rewrite relation, and thus rendering the problem of finding decision procedures for the properties of interest accessible in practice, are characterized in Section 3.3. Section A.1 covers the tree automata foundations of the automated checking of these properties for such theories in the left-linear case and the extension of the Maude SCC tool supporting such checking. Section 3.5 discusses the crucial relationship of constructors for generalized rewrite theories to inductive reasoning for both ground reachability and ground joinability; the use of some of these inductive reachability methods is illustrated with an example in Section 3.6. Section 3.7 discuses related and future work. The proofs omitted in this chapter can be found in Appendix A, including the mechanical proofs for the admissibility of the Maude examples and ITP proof scripts for proving some inductive facts about them.

3.1 Generalized Rewrite Theories

The development in this chapter assumes an order-sorted signature $\Sigma = (S, \leq, F)$, the existence of a subset $K \subseteq S$ of sorts, one per connected component of $(S, \leq)$, and that each operator $f : s_1 \cdots s_n \longrightarrow s$ is also declared at the level of its top sorts $f : k_1 \cdots k_n \longrightarrow k$. The expression F^K denotes the set of overloaded function symbols at the level of top sorts. A Σ-*mapping* χ is a $K^* \times K$-indexed family of function symbols assigning to each $f : k_1 \cdots k_n \longrightarrow k \in F^K$ a finite set $\chi(f) \subseteq \{1, \ldots, n\}$. The *complement* $\overline{\chi}$ of a Σ-mapping χ is the $K^* \times K$-indexed family of function symbols defined for each $f : k_1 \cdots k_n \longrightarrow k \in F^K$ by $\overline{\chi}(f) = \{1, \ldots, n\} \setminus \chi(f)$. The *empty* Σ-*mapping* $\bot$ is defined by $\bot(f) = \varnothing$, for any $f \in F^K$, and the *full* Σ-*mapping* $\top$ is defined by $\top = \overline{\bot}$.

An (unconditional) *generalized rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E, \mu, R, \nu)$ such that (Σ, E, R) is an order-sorted rewrite theory, with unconditional equations E, unconditional rewrite rules R of the form $l \to r$ where $l, r \in T_\Sigma(X)_k$ for some $k \in K$, μ is a Σ-mapping defining for each $f \in F^K$ the positions $\mu(f)$ under which it is *allowed* to perform equational deduction with the equations E, and ν is a Σ-mapping defining for each $f \in F^K$ the positions $\nu(f)$ under which it is *forbidden* to perform rewriting deduction with the rules R. The Σ-mappings μ and ν are called, respectively, the *evaluation strategy* and the *frozenness map* of $\mathcal{R}$. Given a term $t \in T_\Sigma(X)$, the subterm t_π at position π is called *frozen* if and only if there are two positions π_1, π_2 and a natural number n such that $\pi = \pi_1.n.\pi_2$ and $n \in \nu(\pi_1(t))$. The expression $pos_\nu(t)$ denotes the set of *frozen positions* of term t under the frozenness map ν. The occurrence of t_π is called *unfrozen* if and only if it is not frozen.

Similar to ordinary rewrite theories, reasonable executability requirements are needed to make a generalized rewrite theory *admissible* (see Section 2.2.1). It is assumed that the set of Σ-equations of a generalized rewrite theory can be decomposed into a disjoint union $E \cup B$, with B a collection of structural axioms (such as associativity, and/or commutativity, and/or identity) for which there exists a *matching algorithm modulo* B producing a finite number of B-matching solutions, or failing otherwise. It is also assumed that the equations E can be oriented into a set of rewrite rules $\overrightarrow{E}$ inducing a rewrite relation $\to_{E/B}$ that conforms to the evaluation strategy μ and that is ground sort-decreasing, operationally terminating, and confluent modulo B. The set of rewrite rules R is assumed to induce a rewrite relation $\to_{R/B}$ that conforms to the frozenness map ν and is coherent with respect to $\to_{E/B}$. Table 3.1 introduces an inference system, borrowed from [32], that defines the operational semantics of the rewrite relations $\to_{E/B}$ and $\to_{R/B}$ induced by a generalized rewrite theory $\mathcal{R} = (\Sigma, E \cup B, \mu, R, \nu)$.

3.2 Sufficient Completeness and Deadlock Freedom

This section proposes two different notions of constructors and sufficient completeness for a subclass of order-sorted generalized rewrite theories, and further relates these notions to deadlock freedom. In order to focus on the relationship between rewrite constructors and the notion of deadlock freedom, this section considers order-sorted generalized rewrite theories with full strategy map. This assumption helps, mainly, in avoiding involved defi-

$$(R\text{-Ref}) \qquad \frac{t \to^*_{E/B} u}{t \to^*_{R/B} u}$$

$$(R\text{-Trans}) \qquad \frac{t \to^1_{R/B} t' \qquad t' \to^*_{R/B} u}{t \to^*_{R/B} u}$$

$$(R\text{-Cong}) \qquad \frac{t_i \to^1_{R/B} u_i}{f(t_1, \ldots, t_i, \ldots, t_n) \to^1_{R/B} f(t_1, \ldots, u_i, \ldots, t_n)} \quad \text{where } i \notin \nu(f)$$

$$(R\text{-Subs}) \qquad \frac{t \to^*_{E/B} t' \qquad r\theta \to^*_{E/B} u}{t \to^1_{R/B} u} \quad \text{if } t' =_B l\theta \quad \text{where } l \to r \in R$$

$$(E\text{-Ref}) \qquad \frac{\cdot}{t \to^*_{E/B} u} \quad \text{if } t =_A u$$

$$(E\text{-Trans}) \qquad \frac{t \to^1_{E/B} t' \qquad t' \to^*_{E/B} u}{t \to^*_{E/B} u}$$

$$(E\text{-Cong}) \qquad \frac{t_i \to^1_{E/B} u_i}{f(t_1, \ldots, u_i, \ldots, t_n) \to^1_{E/B} f(t_1, \ldots, u_i, \ldots, t_n)} \quad \text{where } i \in \mu(f)$$

$$(E\text{-Subs}) \qquad \frac{\cdot}{t \to^1_{E/B} r\theta} \quad \text{if } t =_B l\theta \quad \text{where } l = r \in E$$

Table 3.1: Operational semantics for a generalized rewrite theory $\mathcal{R} = (\Sigma, E \cup B, \mu, R, \nu)$.

nitions that in the end do not contribute considerably to the understanding of the main idea. On the other hand, Section 3.3 discusses the case in which the theories can have richer equational strategy information.

This section assumes a generalized rewrite theory $\mathcal{R} = (\Sigma, E, \top, R, \nu)$, or simply (Σ, E, R, ν), with order-sorted signature $\Sigma = (S, \leq, F)$. The expression $\mathcal{E}_{\mathcal{R}}$ abbreviates the equational theory (Σ, E).

Definition 2 introduces the basic notion of constructor signature pair.

Definition 2. *A* constructor signature pair *for $\mathcal{R}$ is a pair (Υ, Ω) of order-sorted sub-signatures $\Upsilon = (S, \leq, F_\Upsilon) \subseteq \Omega = (S, \leq, F_\Omega) \subseteq \Sigma$.*

- *The S-sorted set $T_\Omega = \{T_{\Omega,s}\}_{s \in S} \subseteq T_\Sigma$ is called the set of* E-constructor terms.
- *The S-sorted set $C^\Upsilon_{\mathcal{R}} = \{C^\Upsilon_{\mathcal{R},s}\}_{s \in S} \subseteq T_\Omega$ is called the set of* $\mathcal{R}$-constructor terms *and is defined for any $s \in S$ by:*

$$t \in C^\Upsilon_{\mathcal{R},s} \quad \Longleftrightarrow \quad t \in T_{\Omega,s} \wedge pos_{\overline{\nu}}(t) \subseteq pos_\Upsilon(t). \tag{3.1}$$

The intuition behind E-constructor terms is the traditional one, in that any ground Σ-term should be *provably equal* to a term in T_Ω. This is precisely the notion of constructor subsignature already mentioned in Section 2.3. The intuition about $\mathcal{R}$-constructor terms is that any Σ-term should be *rewritable after a finite number of steps* to a term in $C^\Upsilon_{\mathcal{R}}$. Of course, these are *claims* about $\mathcal{R}$ that need to be verified. In particular note that if $\nu = \bot$, then $C^\Upsilon_{\mathcal{R}} = T_\Upsilon$, that is, the $\mathcal{R}$-constructor terms coincide with the Υ-terms. The somewhat subtle point is that, because of frozen positions in some of the operators in Ω, frozen subterms may not be rewritable at all with R, and therefore they may still be Ω-terms and not Υ-terms.

The notion of *sufficient completeness* for $\mathcal{R}$ relative to a constructor signature pair (Υ, Ω) is the expected one, i.e., Ω are the constructors for the equations and Υ the constructor for the rules.

Definition 3 (Sufficient Completeness)**.** *If (Υ, Ω) is a constructor signature pair, then $\mathcal{R}$ is called:*

- E-sufficiently complete *relative to Ω if and only if*

$$(\forall s \in S)(\forall t \in T_{\Sigma,s})(\exists u \in T_{\Omega,s}) \quad \mathcal{E}_{\mathcal{R}} \vdash t = u. \tag{3.2}$$

- $\mathcal{R}$-sufficiently complete *relative to Υ if and only if*

$$(\forall s \in S)(\forall t \in T_{\Sigma,s})(\exists v \in C^\Upsilon_{\mathcal{R},s}) \quad \mathcal{R} \vdash t \to v. \tag{3.3}$$

- Sufficiently complete *relative to* (Υ, Ω) *if and only if statements 3.2 and 3.3 hold.*

The constructors Ω *are called* E-constructors *and the constructors* Υ *are called* $\mathcal{R}$-constructors.

Definition 3 makes explicit use of sort information by requiring the witnesses u and v to have sort less or equal than the sort s of t. This sort requirement can be crucial, for example, when inducting on a variable x_s of sort s. Also note that Definition 3 does not yet make any use of the admissibility assumptions about $\mathcal{R}$. Under such assumptions, the notion of sufficient completeness for $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$ can be further sharpened by relating it to two fundamental sets, namely, the set of terms $Can_{\Sigma,E/B}$ of the canonical term algebra $\mathbf{Can}_{\Sigma,E/B}$ for $(\Sigma, E \cup B)$ and the set $Norm_{\mathcal{R}/B}$ of $\mathcal{R}$-normal forms of $\mathcal{T}_{\mathcal{R}}$.

Definition 4. *Assume* $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$ *is admissible. The* S*-sorted family of sets* $Norm_{\mathcal{R}/A} \subseteq Can_{\Sigma,E/A}$*, called the family of* $\mathcal{R}$-terminal states *of* $\mathbf{Can}_{\mathcal{R}}$ *is defined for each* $s \in S$ *by:*

$$[t]_B \in Norm_{\mathcal{R}/B,s} \iff [t]_B \in Can_{\Sigma,E/B,s} \wedge (\forall u \in T_\Sigma)\mathcal{R} \not\vdash t \rightarrow^1 u. \tag{3.4}$$

Moreover, $\mathcal{R}$ *is called:*

- Ground weakly-normalizing (modulo B) *if and only if*

$$(\forall t \in T_\Sigma)(\exists [v]_B \in Norm_{\mathcal{R}/B})\ \mathcal{R} \vdash t \rightarrow v. \tag{3.5}$$

- Ground sort-decreasing (modulo B) *if and only if*

$$(\forall s \in S)(\forall t \in T_{\Sigma,s})(\forall u \in T_\Sigma)\ \mathcal{R} \vdash t \rightarrow u \implies u \in T_{\Sigma,s} \tag{3.6}$$

Note that the assumption of $\mathcal{R}$ being admissible ensures that the canonical algebra $\mathbf{Can}_{\Sigma,E/B}$ exists and it is well-defined. Also note that notions of ground weak-normalization and sort-decreasingness for $\mathcal{R}$ do not necessarily imply the ground weak-operational termination or sort-decreasingness of the orientable equations E modulo the axioms B.

Theorem 1 gives a sufficient condition for checking sufficient completeness of $\mathcal{R}$ relative to a constructor signature pair under the above-mentioned operational assumptions.

Theorem 1. *Let* (Υ, Ω) *be a constructor signature pair for* $\mathcal{R}$*. If:*

- $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$ *is admissible,*
- $Can_{\Sigma,E/B} \subseteq T_{\Omega,B}$*, and*
- $\mathcal{R}$ *is ground weakly-normalizing and sort-decreasing,*
- $Norm_{\mathcal{R}/B} \subseteq C^{\Upsilon}_{\mathcal{R}/B}$*,*

then $\mathcal{R}$ *is sufficiently complete relative to* (Υ, Ω)*.*

Proof. Let $s \in S$ and $t \in T_{\Sigma,s}$. Let $u = t\downarrow_{E/B}$ and note that u is well-defined because $\mathcal{E}_{\mathcal{R}}$ is admissible by the first assumption. Then $[u]_B \in Can_{\Sigma,E/B}$, $t =_{E/B} u$, and (by ground sort-decreasingness of $\mathcal{E}_{\mathcal{R}}$) u has sort s. From the second assumption it follows that u witnesses Statement 3.2, and thus $\mathcal{R}$ is E-sufficiently complete relative to Ω. On the other hand, $\mathcal{R}$ being ground weakly-normalizing and sort-decreasing (third assumption), implies the existence of $[v]_B \in Norm_{\mathcal{R}/B}, s$ such that $\mathcal{R} \vdash t \to v$. Then, by the fourth assumption there is $v' \in C^{\Upsilon}_{\mathcal{R},s}$ satisfying $[v]_B = [v']_B$ and such that $\mathcal{R} \vdash t \to v'$, that is, v' is a witness for Statement 3.3. This implies $\mathcal{R}$-sufficient completeness relative to Υ. Therefore, $\mathcal{R}$ is sufficiently complete relative to (Υ, Ω). □

Definition 5. *If* (Υ, Ω) *is a constructor signature pair for* $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$*, then* $\mathcal{R}$ *is called* canonically sufficiently complete *relative to* (Υ, Ω) *if and only if it satisfies the premises in Theorem 1. Furthermore,*

- Ω *is called a* signature of E-free constructors modulo B *if and only if*

$$Can_{\Omega,E/B} = T_{\Omega/B}. \tag{3.7}$$

- Υ *is called a* signature of $\mathcal{R}$-terminal constructors *if and only*

$$Norm_{\mathcal{R}/B} = C^{\Upsilon}_{\mathcal{R}/B}. \tag{3.8}$$

Condition 3.8 exactly means that $\mathcal{R}$ is *deadlock free* outside $C^{\Upsilon}_{\mathcal{R}/B}$. Therefore, if $\mathcal{R}$ is canonically sufficiently complete relative to (Υ, Ω), then it is deadlock free outside $C^{\Upsilon}_{\mathcal{R}/B}$. The S-sorted sets $T_{\Omega/B}$ and $C^{\Upsilon}_{\mathcal{R}/B}$ provide respective *envelopes* containing the key sets $Can_{\Sigma,E/B}$ (the set of states of $\mathbf{Can}_{\mathcal{R}}$) and $Norm_{\mathcal{R}/B}$ (the set of terminal states of $\mathbf{Can}_{\mathcal{R}}$). Furthermore, if Ω is a signature of E-free constructors modulo B, and Υ is a signature of $\mathcal{R}$-terminal constructors, these envelopes are *tight*, in the sense that $T_{\Omega/B}$ and $T_{\Upsilon/B}$ *exactly characterize* $Can_{\Sigma,E/B}$ and $Norm_{\mathcal{R}/B}$, respectively. Figure 3.1 depicts the containment relationships between these S-sorted sets of terms.

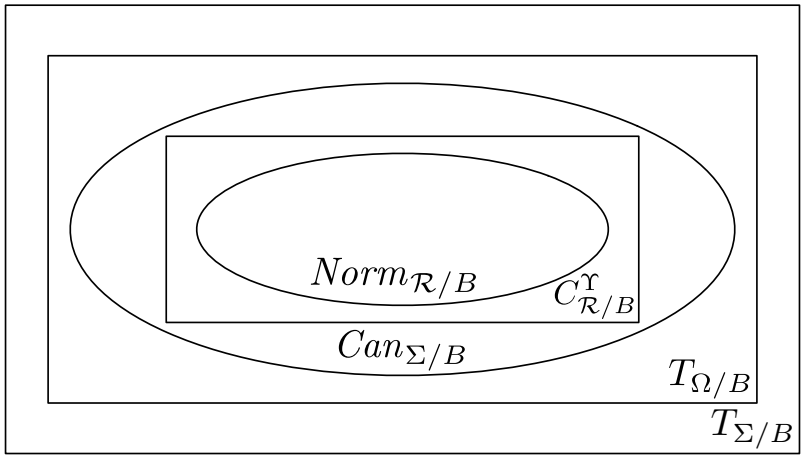

Figure 3.1: Containment relationships between some sets of terms related to generalized rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$ with constructor signature pair (Υ, Ω) that is canonically sufficiently complete.

3.3 Checking Canonical Sufficient Completeness

For purposes of checking sufficient completeness of a generalized rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$, it is helpful to find simple conditions under which the rewrite relations $\rightarrow_{E/B}$ and $\rightarrow_{R/B}$ can be jointly captured by a single rewrite relation. This is not entirely straightforward, because evaluation strategies apply to E but not necessarily to R and frozenness requirements apply to R but not necessarily to E. The goal in this section is to prove that for the purpose of checking the canonical sufficient completeness of $\mathcal{R}$, it is correct to reason about the rewrite relation $\rightarrow_{E \cup R/B}$, which is simpler than $\rightarrow_{R/B}$ for this purpose. This section assumes an admissible order-sorted rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$ with signature $\Sigma = (S, \leq, F)$.

The key insight is that the complement $\overline{\mu}$ of the evaluation strategy μ can sometimes be seen as a frozenness map without altering the overall reachability properties of the original generalized rewrite theory $\mathcal{R}$. As explained in [54], under appropriate admissibility conditions, the notion of canonical term algebra $\mathbf{Can}_{\Sigma,E/B}$ can be relativized to a map μ specifying E-reducible positions, yielding an algebra $\mathbf{Can}^{\mu}_{\Sigma,E/B}$. The way to combine $\rightarrow_{E/B}$ and $\rightarrow_{R/B}$ into a single rewrite relation $\rightarrow_{E \cup R/B}$ without changing the mathematical semantics of the given theory $\mathcal{R}$ is then, in essence: (i) to require that $\overline{\mu} = \nu$, so that $\rightarrow_{E/B}$ and $\rightarrow_{R/B}$ obey the same frozenness constraints, and (ii) to further require that the canonical term algebra remains unchanged. This is captured by the notion of *simple generalized rewrite theories*, a subclass of generalized rewrite theories for which is sound and

complete, relative to reachability analysis, to ignore the semantic distinction between equations and rules at the operational level, even in the presence of strategy and frozenness information for both equations and rules.

Definition 6 (Simple Generalized Rewrite Theory)**.** *The generalized rewrite theory $\mathcal{R}$ is called* simple *if and only* $\mathbf{Can}_{\Sigma,E/B} \simeq \mathbf{Can}^{\mu}_{\Sigma,E/B}$*, where* $\mu = \overline{\nu}$*.*

Example 1 illustrates the notion of simple generalized rewrite theories in Definition 6.

Example 1. *Consider the specification* `BAG-CHOICE+CARD`*, representing a generalized rewrite theory*

$$(\Sigma^{\mathtt{BCC}}, E^{\mathtt{BCC}} \cup B^{\mathtt{BCC}}, \mu^{\mathtt{BCC}}, R^{\mathtt{BCC}}, \nu^{\mathtt{BCC}})$$

in the language of Maude, that models bags (or multisets) of natural numbers in Peano notation:

```
mod BAG-CHOICE+CARD is
  sorts Nat .
  op 0  : -> Nat [ctor metadata "rctor"] .
  op s_ : Nat -> Nat [ctor metadata "rctor"] .

  sorts NeBag Bag .
  subsort Nat < NeBag < Bag .
  op mt  : -> Bag [ctor metadata "rctor"] .
  op __  : Bag Bag -> Bag [assoc comm id: mt ctor] .
  op |_| : Bag -> Nat [strat(0) frozen(1)] .

  eq [card0]  :
     | mt |
   = 0 .
  eq [card1]  :
     | N:Nat B:Bag |
   = s | B:Bag | .

  rl [choice] :
     N:Nat NeB:NeBag
  => N:Nat .
endm
```

The equational constructor `ctor` *and* rewrite constructor `metadata "rctor"` *declarations define the constructor signature pair for* `BAG-CHOICE+CARD`*. Equations* `[card0]` *and* `[card1]` *fully define the cardinality of any bag of natural numbers. Rule* `[choice]` *non-deterministically chooses an element of a non-empty bag of natural numbers. In Maude, an evaluation strategy μ for the equations is declared with the attribute keyword* `strat`*, which always begins with a* 0 *and is followed by the numbers $i_1, \ldots, i_m$ such that*

$\mu(f) = \{i_1, \ldots, i_m\}$. *Instead, a frozenness mapping* ν *for the rewrite rules is declared with the attribute keyword* `frozen` *(see [23] for details). In particular, for this specification, the evaluation strategy* $\mu^{\mathtt{BCC}}$ *and the frozenness map* $\nu^{\mathtt{BCC}}$ *satisfy:*

$$\mu^{\mathtt{BCC}}(|\_|) = \bot = \nu^{\mathtt{BCC}}(|\_|)$$
$$\mu^{\mathtt{BCC}}(f) = \overline{\bot} = \nu^{\mathtt{BCC}}(f), \text{ for any } f \in \{0, s, mt, \_\_\}.$$

The frozenness map $\nu^{\mathtt{BCC}}$ *prevents the rewrite rule* `[choice]` *from performing any rewrites below any occurrence of the cardinality function symbol* $|\_|$ *in any* $\Sigma^{\mathtt{BCC}}$*-term, which is necessary to avoid the cardinality of a bag itself to be rewritten to a smaller cardinality. Observe that* $\overline{\mu^{\mathtt{BCC}}} = \nu^{\mathtt{BCC}}$. *Furthermore, whether* $\mathbf{Can}_{\Sigma^{\mathtt{BCC}}, E^{\mathtt{BCC}}/B^{\mathtt{BCC}}}$ *and* $\mathbf{Can}^{\mu^{\mathtt{BCC}}}_{\Sigma^{\mathtt{BCC}}, E^{\mathtt{BCC}}/B^{\mathtt{BCC}}}$ *coincide is a decidable property [54] that can be automatically checked by Maude's SCC tool for this specification. Moreover, the admissibility of this specification can also be checked automatically. See Appendix A for the mechanical proofs. In this case,* `BAG-CHOICE+CARD` *is indeed a simple generalized rewrite theory.*

As an important remark, observe that any generalized rewrite theory $(\Sigma, E \cup B, \mu, R, \nu)$ is inherently a simple generalized rewrite theory when ν and μ are ignored.

Definition 7. *For any generalized rewrite theory* $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$, *define* $\mathcal{R}_E = (\Sigma, B, \overrightarrow{E}, \nu)$ *and* $\mathcal{R}_{E \cup R} = (\Sigma, B, \overrightarrow{E} \cup R, \nu)$.

Although $\mathcal{R}_E$ and $\mathcal{R}_{E \cup R}$ ignore at the operational level the semantic distinction between the equations E and the rules R of $\mathcal{R}$, these two simple generalized rewrite theories are sound and complete relative to reachability analysis with respect to $\mathcal{R}$, even in the presence of strategy and frozenness information for the equations E and the rules R, respectively (see Definition 6). The key observations are that under some conditions: (i) the sets $Can_{\Sigma,E/B}$ and $Norm_{\mathcal{R}_E/B}$ coincide, and (ii) the sets $Norm_{\mathcal{R}_{E\cup R}/B}$ and $Norm_{\mathcal{R}/B}$ also coincide, even though $\mathcal{R}_{E\cup R}$ has a simpler rewrite relation than $\mathcal{R}$. These claims are verified in Theorem 2.

Theorem 2. *If* $\mathcal{R}$ *is an admissible simple generalized rewrite theory, then the following equalities hold:*

1. $Norm_{\mathcal{R}_E/B} = Can_{\Sigma,E/B}$.
2. $Norm_{\mathcal{R}_{E\cup R}/B} = Norm_{\mathcal{R}/B}$.

Proof. For (1) observe that since $\mathcal{R}$ is admissible and it is a simple generalized rewrite theory, it follows that $Norm_{\mathcal{R}_E/B} = Can^{\overline{\nu}}_{\Sigma,E/B} = Can_{\Sigma,E/B}$. For (2) first note that $Norm_{\mathcal{R}_{E\cup R}/B} \subseteq Norm_{\mathcal{R}_E/B}$ because $\overrightarrow{E} \subseteq \overrightarrow{E \cup R}$ and then, from (1), $Norm_{\mathcal{R}_{E\cup R}/B} \subseteq Can_{\Sigma,E/B}$. Let $s \in S$ and $t \in T_{\Sigma,s}$, and observe:

$$
\begin{array}{ll}
& [t]_B \in Norm_{\mathcal{R}_{E\cup R}/B,s} \\
\iff & \quad \{ \text{ by definition of } Norm_{\mathcal{R}_{E\cup R}/B,s} \} \\
& [t]_B \in Can_{\Sigma,E/B,s} \ \wedge \ (\forall u \in T_\Sigma)\mathcal{R}_{E\cup R} \not\vdash t \rightarrow^1 u \\
\iff & \quad \{ \text{ by ground coherence of } R \text{ w.r.t. } E \text{ modulo } B \} \\
& [t]_B \in Can_{\Sigma,E/B,s} \ \wedge \ (\forall u \in T_\Sigma)\mathcal{R} \not\vdash t \rightarrow^1 u \\
\iff & \quad \{ \text{ by definition of } Norm_{\mathcal{R}/B,s} \} \\
& [t]_B \in Norm_{\mathcal{R}/B,s}.
\end{array}
$$

□

3.4 Decision Procedures with Propositional Tree Automata

Given a constructor signature pair (Υ, Ω) for an simple generalized rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$, this section presents sufficient conditions under which the problems of deciding whether: (i) $\mathcal{R}$ is canonically sufficiently complete relative to (Υ, Ω), (ii) Ω is a signature of E-free constructors modulo B, and (iii) Υ is signature of $\mathcal{R}$-terminal constructors, can all be reduced to emptiness checks of languages recognized by propositional tree automata. The treatment here generalizes that of [54, 52], where such automata were used to check E-sufficient completeness of order-sorted (equational) specifications.

Tree automata techniques have been used to check the sufficient completeness of equational specifications, e.g., [27, 54, 52]. Propositional Tree Automata [55] (PTA) extend traditional equational tree automata by allowing inputs to range over a many-kinded signature instead of over an unsorted signature, recognition is done modulo axioms, and an input term is accepted if its set of reachable states satisfies a given proposition.

Definition 8. *A* propositional tree automaton *(PTA) is a tuple $\mathcal{B}$ of the form $(K, F, Q, \Gamma, B, \Delta)$ where*

- *(K, F) is a many-kinded signature, i.e., a set K of kinds and a $K^* \times K$-indexed set F of function symbols,*

- $Q = \{Q_k\}_{k \in K}$ *is a* K*-indexed set of pairwise disjoint sets of states such that* $Q_k \cap F_{\epsilon,k'} = \varnothing$ *for each* $k, k' \in K$,
- $\Gamma = \{\gamma_k\}_{k \in K}$ *is a* K*-indexed set of Boolean propositions where the atoms in each* γ_k *are the states in* Q_k,
- B *is a set of unconditional* (K, F)*-equational axioms, and*
- Δ *is a set of* transition rules *of the form* $f(p_1, \ \ldots, p_n) \to q$, *or* $p \to q$, *for some* $k \in K$, $p, q \in Q_k$, $f \in F_{k_1 \ldots k_n, k}$, *and each* $p_i \in Q_{k_i}$.

A PTA $\mathcal{B}$ can be regarded as a rewrite theory $\mathcal{R}_{\mathcal{B}}$, so that $L(\mathcal{B})$, the *language accepted* by $\mathcal{B}$, can be defined in terms of reachability in $\mathcal{R}_{\mathcal{B}}$.

Definition 9. *Let* $\mathcal{B} = (K, F, Q, \Gamma, B, \Delta)$ *be a PTA and let* $\Sigma = (K, \varnothing, F \cup Q)$, *where each* $q \in Q_k$ *is viewed as a constant of kind* $k \in K$. *Then,* $\mathcal{R}_{\mathcal{B}} = (\Sigma, B, \Delta)$ *is the* associated rewrite theory *of* $\mathcal{B}$ *and the* move relation $\to_{\mathcal{B}}$ *is the binary relation defined for* $t, u \in T_\Sigma$ *by:*

$$t \to_{\mathcal{B}} u \quad \iff \quad t \to^1_{\mathcal{R}_{\mathcal{B}}} u.$$

For each $k \in K$, *let* $reach_{\mathcal{B},k} : T_\Sigma \longrightarrow \mathcal{P}(Q_k)$ *be the map defined by:*

$$t \mapsto \{q \in Q_k \mid t \to^*_{\mathcal{B}} q\}.$$

Then, $L(\mathcal{B}) = \{L(\mathcal{B})_k\}_{k \in K}$, *where*

$$L(\mathcal{B})_k = \{t \in T_{\Sigma,k} \mid reach_{\mathcal{B},k}(t) \models \gamma_k\},$$

and $\models$ *denotes the satisfaction relation of propositional logic.*

When the emptiness problem for PTA is decidable, other typical decision problems, such as inclusion, universality and intersection-emptiness are all decidable due to the Boolean closure properties of PTAs. As shown in [55], when B is any combination of associativity, commutativity and identity axioms, but excluding the case in which there is an associative but not commutative symbol in B, the emptiness problem for PTA is decidable. In the special case in which there are associative but not commutative symbols in B, machine learning techniques can be applied to create a semi-decision procedure which can always show non-emptiness, and can show emptiness under certain regularity conditions [55].

Definition 10. *A simple generalized rewrite theory* $\mathcal{R} = (\Sigma, B, R, \nu)$ *is* PTA-checkable *if and only if*

- $\mathcal{R}$ *is ground weakly-normalizing and ground sort-decreasing,*
- $S_k \cap F_{\epsilon,k} = \varnothing$ *for each* $k \in K$*,*
- *the axioms* B *are any combination of associativity, commutativity and identity axioms, except for the cases in which a symbol is associative but not commutative, and*
- *every rule in* R *is of the form* $f(t_1, \ldots, t_n) \to t$*, with* $f(t_1, \ldots, t_n)$ *linear.*

3.4.1 Checking Sufficient Completeness

An executable simple generalized rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$, with signature $\Sigma = (S, \leq, F)$, is *not* canonically sufficiently complete relative to the constructor signature pair (Υ, Ω) if and only if there is a sort $s \in S$ and a term $t \in T_{\Sigma,s}$ such that either

(i) $[t]_B \in \mathit{Norm}_{\mathcal{R}_E/B,s} \cap \left(T_{\Sigma/B,s} \setminus T_{\Omega/B,s}\right)$ or

(ii) $[t]_B \in \mathit{Norm}_{\mathcal{R}_{E\cup R}/B,s} \cap \left(T_{\Sigma/B,s} \setminus C^{\Upsilon}_{\mathcal{R}/B,s}\right)$.

Under PTA-checkability, canonical sufficient completeness can be reduced to an emptiness problem of PTAs by constructing two automata that accept precisely those terms $t \in T_{\Sigma,s}$ such that $[t]_B$ satisfies (i) or (ii).

Theorem 3. *Let* $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$ *be an admissible, ground weakly-normalizing, and ground sort-decreasing simple generalized rewrite theory, and let* (Υ, Ω) *be a constructor signature pair for* $\mathcal{R}$*. If* $\mathcal{R}_E$ *and* $\mathcal{R}_{E\cup R}$ *are PTA-checkable, then there are PTAs* $\mathcal{B}_E$ *and* $\mathcal{B}_{E\cup R}$ *such that* $\mathcal{R}$ *is canonically sufficiently complete relative to* (Ω, Υ) *if and only if* $L(\mathcal{B}_E) \cup L(\mathcal{B}_{E\cup R}) = \varnothing$*.*

Proof. See Section A.1 in Appendix A. □

3.4.2 Checking E-free and $\mathcal{R}$-terminal Constructors

If a simple generalized rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$, with signature $\Sigma = (S, \leq, F)$, is canonically sufficiently complete relative to (Υ, Ω), then:

(i) Ω is an E-free constructor signature if and only if

$$(\forall s \in S)\ T_{\Omega/B,s} \setminus \mathit{Norm}_{\mathcal{R}_E/B,s} = \varnothing.$$

(ii) Υ is an $\mathcal{R}$-terminal constructor signature if and only if

$$(\forall s \in S)\ C^{\Upsilon}_{\mathcal{R}/B,s} \setminus Norm_{\mathcal{R}/B,s} = \varnothing.$$

Theorem 4. *Let $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$ be a simple generalized rewrite theory that canonically sufficiently complete relative to the constructor signature pair (Ω, Υ). If $\mathcal{R}_E$ and $\mathcal{R}_{E\cup R}$ are PTA-checkable, then there are PTAs $\mathcal{F}_E$ and $\mathcal{D}_{E\cup R}$ such that Ω is a signature of E-free constructors modulo B if and only if $L(\mathcal{F}_E) = \varnothing$, and Υ is signature of $\mathcal{R}$-deadlock constructors if and only if $L(\mathcal{D}_{E\cup R}) = \varnothing$.*

Proof. See Section A.1 in Appendix A. □

3.4.3 The Extended Maude Sufficient Completeness Checker

The Maude Sufficient Completeness Checker [54] (SCC) has been extended to construct the automata defined in the proofs of Theorem 3 and Theorem 4, so that sufficient completeness checks, and also checks for E-free constructors and $\mathcal{R}$-terminal constructors, can be automatically handled for such generalized rewrite theories.

Given an admissible simple generalized rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$ annotated with constructor signature pair (Υ, Ω) in the syntax of Maude and satisfying the conditions in Theorem 3, the SCC's command `scc-df` builds the automata $\mathcal{B}_E$ and $\mathcal{B}_{E\cup R}$ and checks for their emptiness. For the example in Section 3.3, it works as expected:

```
Maude> (scc-df BAG-CHOICE+CARD .)
Checking sufficient completeness and deadlock freeness of BAG-CHOICE+CARD...
Success: The equational subtheory of BAG-CHOICE+CARD is sufficiently complete
  under the assumption that it is ground weakly-normalizing, ground confluent,
  and ground sort-decreasing.
Success: The rewrite theory BAG-CHOICE+CARD is deadlock-free outside rctor-terms
  under the assumption that it is ground weakly-normalizing, ground sort-decreasing,
  and ground coherent.
```

For $\mathcal{R}$ and (Υ, Ω) as above, and under the assumption of $\mathcal{R}$ being canonically sufficiently complete relative to (Υ, Ω), the Sufficient Completeness Checker commands `free-terminal` builds the automata $\mathcal{F}_E$ and $\mathcal{G}_{E\cup R}$ and checks for their emptiness.

```
Maude> (free-terminal BAG-CHOICE+CARD .)
Checking freeness of constructors of BAG-CHOICE+CARD...
Success: The equational subtheory of BAG-CHOICE+CARD has equational free
  constructors under the assumption that it is sufficiently complete, ground
```

```
  weakly-normalizing, ground confluent, and ground sort-decreasing.
Success: BAG-CHOICE+CARD has terminal constructors under the assumption that it
  is deadlock-free outside rctor-terms, ground weakly-normalizing, ground
  sort-decreasing, and ground coherent.
```

As an additional example, consider the one-sorted simple generalized rewrite theory `NAT-LIST` adapted from [45]:

```
mod NAT-LIST is
 sort List .
 ops 0 nil err          : -> List           [ctor metadata "rctor"] .
 ops nats zeros         : -> List           [ctor] .
 op  s                  : List -> List      [ctor metadata "rctor"] .
 ops incr adx head tail : List -> List      [ctor] .
 op  cons               : List List -> List [ctor metadata "rctor"] .
 vars X L : List .
 rl incr(nil) => nil .
 rl incr(cons(X,L)) => cons(s(X),incr(L)) .
 rl adx(nil) => nil .
 rl adx(cons(X,L)) => incr(cons(X,adx(L))) .
 rl nats => adx(zeros) .
 rl zeros => cons(0,zeros) .
 rl zeros => cons(0,nil) .
 rl head(cons(X,L)) => X .
 rl tail(cons(X,L)) => L .
 rl adx(0) => err .
 rl adx(s(X)) => err .
 rl incr(0) => err .
 rl incr(s(X)) => err .
 rl tail(nil) => nil .
 rl tail(0) => err .
 rl tail(s(X)) => err .
 rl head(nil) => nil .
 rl head(0) => err .
 rl head(s(X)) => err .
endm
```

Running the `scc-df` command on `NAT-LIST` yields and error:

```
Maude> (scc-df NAT-LIST .)
Checking sufficient completeness and deadlock freeness of NAT-LIST...
Success: The equational subtheory of NAT-LIST is sufficiently complete under
  the assumption that it is ground weakly-normalizing, ground confluent,
  and ground sort-decreasing.
Failure: The term adx(err) is a terminal term outside rctor-terms of sort List.
```

It turns out that the rewrite system given in [45] was missing rewrite rules for defining `adx`, `incr`, `tail`, and `head` when the argument was the error list `err`. By adding the rewrite rules "`adx(err) => err`", "`incr(err) => err`", "`tail(err) => err`", and "`head(err) => err`" to complete the above Maude specification to one called `NAT-LIST-COMPLETE`, the sufficient completeness check succeeds.

```
Maude> (scc-df NAT-LIST-COMPLETE .)
Checking sufficient completeness and deadlock freeness of NAT-LIST-COMPLETE...
Success: The equational subtheory of NAT-LIST-COMPLETE is sufficiently complete
  under the assumption that it is ground weakly-normalizing, ground confluent,
  and ground sort-decreasing.
Success: The rewrite theory NAT-LIST-COMPLETE is deadlock-free outside rctor-terms
  under the assumption that it is ground weakly-normalizing, ground sort-decreasing,
  and ground coherent.
```

3.5 Constructor-Based Reachability Analysis

This section discusses the role that $\mathcal{R}$-constructors and E-constructors can play in inductive proofs of ground reachability and ground joinability properties for a rewrite theory $\mathcal{R}$. The discussion here does *not* cover in detail either the theoretical foundations for the soundness of the inductive arguments given in the examples, or the alternative proof techniques that could be used for proving ground reachability and ground joinability properties. The aim here is more modest, namely, to characterize when it is sound to use constructors in such inductive reachability proofs. A simple example in Section 3.6 illustrates the key role constructors can play in such proofs.

3.5.1 Ground Reachability

Ground reachability is an inductive property of rewrite theories. In particular, it is important for establishing reachability properties of concurrent systems specified by generalized rewrite theories, for instance, when algorithmic model checking techniques are limited. This section clarifies the role of constructors in ground reachability proofs.

Definition 11. *Let $\mathcal{R}$ be a simple generalized rewrite theory with signature $\Sigma = (S, \leq, F)$ and let $t, u \in T_\Sigma(X)_s$, for some $s \in S$. The term u is* ground $\mathcal{R}$-reachable *from t, written $\mathcal{R} \Vdash t \to u$, if and only if*

$$(\forall \sigma : X \longrightarrow T_\Sigma)\ \mathcal{R} \vdash t\sigma \to u\sigma. \tag{3.9}$$

In general, reasoning in $\mathcal{R}$ about an inductive property φ requires a deduction relation $\vdash_{ind}$ with inductive inference support (and more powerful than $\vdash$) such that if $\mathcal{R} \vdash_{ind} \varphi$ then $\mathcal{R} \Vdash \varphi$. Note that there is no hope for the converse in general to hold because of Gödel's *Incompleteness Theorem*. The relationship between these relations is made explicit by the following

chain of (meta) implications:

$$\mathcal{R} \vdash \varphi \quad \Longrightarrow \quad \mathcal{R} \vdash_{ind} \varphi \quad \Longrightarrow \quad \mathcal{R} \Vdash \varphi \quad \Longleftrightarrow \quad \mathcal{T}_{\mathcal{R}} \models \varphi.$$

The notion of $\mathcal{R} = (\Sigma, E, R, \nu)$ being sufficiently complete relative to a constructor signature pair (Υ, Ω) is important for inductive reasoning at both equational and rewrite levels because different explicit or implicit inductive proof methods (including structural induction) become much more effective when the inductions can be restricted to constructor terms.

Inductive reasoning about rewrite sequents for a rewrite theory $\mathcal{R}$ is somewhat subtle, particularly in the presence of frozenness information. The first subtle point is that, because of frozen variables occurring in some of the terms in the goal, it is wrong to assume that it is enough to "instantiate" the induction variables with $\mathcal{R}$-constructor terms. The problem is that frozen subterms may be Ω-terms and not $\mathcal{R}$-constructor terms, and so they may not be rewritable at all with R into $\mathcal{R}$-constructor terms. The solution is then to consider the possibility of frozen subterms being Ω-terms. For ground reachability the reasoning is even subtler, because even if frozen variables are handled as just explained, there is the hidden risk of the target term in the reachability goal becoming a "moving" target. The problem here is that rewrite sequents are not symmetric and the proofs obtained by induction over $\mathcal{R}$-constructor terms in general cannot be "lifted" to proofs for Ω-terms. A solution for this problem is to require that the E-equivalence class of the target term in the reachability goal be invariant under the substitutions used in the proof, by imposing a simple condition on the variables occurring in the target term. As a direct consequence of these observations, for a constructor-based structural induction proof of ground reachability to be sound, it is mandatory to consider all E-constructor terms of the given sort s not only when inducting on a frozen variable x_s, but also when this variable (even if not frozen) occurs in the target term of the reachability goal.

Theorem 5. *Let $\mathcal{R}$ be a simple generalized rewrite theory with signature $\Sigma = (S, \leq, F)$ and frozenness map ν, let $t, u \in T_\Sigma(X)_s$ for some $s \in S$, and let $\theta : X \longrightarrow T_\Sigma$. If $\mathcal{R}$ is sufficiently complete relative to the constructor signature pair (Υ, Ω), then there exists $\eta : X \longrightarrow T_\Omega$ such that:*

1. *$\eta(x) \in C_{\mathcal{R}}^{\Upsilon}$ for $x \in \overline{\nu}(t) \setminus vars(u)$, and $\mathcal{E}_{\mathcal{R}} \vdash \theta(x) = \eta(x)$ for $x \in vars(u)$,*

2. *$\mathcal{R} \vdash t\theta \rightarrow t\eta$.*

Furthermore, $\mathcal{R} \Vdash t \to u$ if and only if $\mathcal{R} \vdash t\eta \to u\eta$ for each η as above.

Proof. (Existence of η) Let $\theta : X \longrightarrow T_\Sigma$ be as given. Since $\mathcal{R}$ is sufficiently complete relative to (Υ, Ω), there are $\beta : X \longrightarrow T_\Omega$ s.t. $\mathcal{E}_\mathcal{R} \vdash \theta(x_{s_i}) = \beta(x_{s_i})$, and $\rho : X \longrightarrow C_\mathcal{R}^\Upsilon$ s.t. $\mathcal{R} \vdash \beta(x_{s_i}) \to \rho(x_{s_i})$, for each $x_{s_i} \in X$. Take $\eta : X \longrightarrow T_\Omega$ to be the map $x \mapsto \rho(x)$ if $x \in \overline{\nu}(t) \setminus vars(u)$, and $x \mapsto \beta(x)$ otherwise. Observe that η fulfills (1) and (2). ($\Longrightarrow$) Let $\eta : X \longrightarrow T_\Omega$ satisfy (1) and (2). Observe that $C_\mathcal{R}^\Upsilon \subseteq T_\Omega \subseteq T_\Sigma$, and hence $\mathcal{R} \vdash t\eta \to u\eta$ follows from the assumption. ($\Longleftarrow$) Let $\theta : X \longrightarrow T_\Sigma$. There exists $\eta : X \longrightarrow T_\Omega$ satisfying conditions (1) and (2), and such that $\mathcal{R} \vdash t\eta \to u\eta$ from the assumption. Then, $\mathcal{R} \vdash t\theta \to u\eta$ follows from the transitivity of rewrite sequents. Observe that η is such that $\mathcal{E}_\mathcal{R} \vdash u\eta = u\theta$, and then $\mathcal{R} \vdash t\theta \to u\theta$. Therefore $\mathcal{R} \Vdash t \to u$, as desired. □

3.5.2 Ground Joinability

The notion of ground joinability is of great importance in the field of term rewriting and also in theorem proving, see, e.g., [83, 6, 60, 69, 9, 2, 15]. In particular, it is a key technique for proving ground confluence. This section explains how $\mathcal{R}$-constructors can be used to prove ground joinability and illustrate the ideas by means of a simple example. A detailed discussion of alternative proof techniques for ground reachability is outside the scope of this paper: the focus here is in clarifying the role that $\mathcal{R}$-constructors can play in such proofs.

Definition 12. *Let $\mathcal{R}$ be a simple generalized rewrite theory, with signature $\Sigma = (S, \leq, F)$, and let $t, u \in T_\Sigma(X)_s$, for some $s \in S$. The terms t and u are called* ground $\mathcal{R}$-joinable*, written $\mathcal{R} \Vdash t \downarrow u$, if and only if*

$$(\forall \sigma : X \longrightarrow T_\Sigma)\ \mathcal{R} \vdash t\sigma \downarrow u\sigma. \tag{3.10}$$

The notion of $\mathcal{R} = (\Sigma, E, R, \nu)$ being sufficiently complete relative to a constructor signature pair (Υ, Ω) is also important for ground joinability. For inductive reasoning about ground joinability the *only* subtle point is that of frozen variables occurring in some of the terms in the goal: it is wrong to assume that it is enough to "instantiate" the induction variables with $\mathcal{R}$-constructor terms. As explained for ground reachability, the problem is that frozen subterms may be Ω-terms and not $\mathcal{R}$-constructor terms, and so they may not be rewritable at all with R into $\mathcal{R}$-constructor terms. As shown by Theorem 6, it is sufficient to consider the possibility of frozen subterms being

Ω-terms for inductive proofs to be sound. That is, for a constructor-based structural induction proof of ground joinability to be sound, it is sufficient to consider all equational constructor terms of sort s when inducting on a frozen variable x_s.

Theorem 6. *Let $\mathcal{R}$ be a simple generalized rewrite theory, with signature $\Sigma = (S, \leq, F)$ and frozenness map ν, and let $t, u \in T_\Sigma(X)_s$, for some $s \in S$. If $\mathcal{R}$ is sufficiently complete relative to the constructor signature pair (Υ, Ω), then $\mathcal{R} \Vdash t \downarrow u$ if and only if $\mathcal{R} \vdash t\eta \downarrow u\eta$ for each $\eta : X \longrightarrow T_\Omega$ such that if $\eta(x) \in C^\Upsilon_\mathcal{R}$ then $x \in \overline{\nu}(t, u)$, for all $x \in X$.*

Proof. ($\Longrightarrow$) Let η be such that if $\eta(x) \in C^\Upsilon_\mathcal{R}$ then $x \in \overline{\nu}(t, u)$ for each $x \in X$. Observe that $C^\Upsilon_\mathcal{R} \subseteq T_\Omega \subseteq T_\Sigma$, and hence $\mathcal{R} \vdash t\eta \rightarrow u\eta$ follows from the assumption. ($\Longleftarrow$) Let $\theta : X \longrightarrow T_\Sigma$. Since $\mathcal{R}$ is sufficiently complete relative to (Υ, Ω), there are $\beta : X \longrightarrow T_\Omega$ s.t. $\mathcal{E}_\mathcal{R} \vdash \theta(x_{s_i}) = \beta(x_{s_i})$, and $\rho : X \longrightarrow C^\Upsilon_\mathcal{R}$ s.t. $\mathcal{R} \vdash \beta(x_{s_i}) \rightarrow \rho(x_{s_i})$, for each $x_{s_i} \in X$. Take $\eta : X \longrightarrow T_\Omega$ to be the map $x \mapsto \rho(x)$ if $x \in \overline{\nu}(t, u)$ and $x \mapsto \beta(x)$ otherwise. Observe that $\mathcal{R} \vdash t\theta \rightarrow t\eta$ and $\mathcal{R} \vdash u\theta \rightarrow u\eta$, and η is defined so that $\mathcal{R} \vdash t\eta \downarrow u\eta$ follows from the assumptions. Therefore $\mathcal{R} \Vdash t \downarrow u$, as desired. □

3.6 Formal Properties of `CHANNEL`

This section shows the use of the constructor-based inductive techniques introduced in sections 3.5.1 and 3.5.2 for proving reachability properties of rewrite theories based on constructors, with an example of a concurrent system communicating through a channel. This example also illustrates how the PTA-based decision procedures defined in Section A.1 can automatically aid, not only in proving the sufficient completeness and deadlock freedom of the specification, but also in obtaining proofs of other inductive reachability properties. Some mechanical proofs are shown in the development of the section; the remaining ones can be found in Appendix A.

Consider the specification `CHANNEL` representing a generalized rewrite theory

$$(\Sigma^{\texttt{CHANNEL}}, E^{\texttt{CHANNEL}} \cup B^{\texttt{CHANNEL}}, R^{\texttt{CHANNEL}})$$

in the language of Maude. It models a system comprising a sender of a list of numbers, a receiver of such a list, and a communication channel through which numbers are sent to the receiver, and acknowledgments are sent back to the sender.

```
mod CHANNEL is
  sorts Nat List NilList EmptyChannel Channel Terminal State .
  subsorts NilList < List .
  subsorts Nat EmptyChannel < Channel .
  subsorts Terminal < State .

  op 0  : -> Nat [ctor metadata "rctor"] .
  op s_ : Nat -> Nat [ctor metadata "rctor"] .

  op nil : -> NilList [ctor metadata "rctor"] .
  op _;_ : Nat List -> List [ctor metadata "rctor"] .
  op _@_ : List List -> List .
  op mt  : -> EmptyChannel [ctor metadata "rctor"] .
  op ack : -> Channel [ctor metadata "rctor"] .

  op <_:_:_> : List Channel List -> State [ctor] .
  op <_:_:_> : NilList EmptyChannel List
               -> Terminal [ctor metadata "rctor"] .

  vars M N : Nat .  vars L L' : List .

  eq [ap01] : nil @ L = L .
  eq [ap02] : (N ; L) @ L' = N ; (L @ L') .
  rl [send] : < N ; L : mt : L' > => < L : N : L' > .
  rl [recv] : < L : N : L' > => < L : ack : L' @ (N ; nil) > .
  rl [ack]  : < L : ack : L' > => < L : mt : L' > .
endm
```

States of this systems are (ground) terms of sort `State`, that is, ground terms of the form

```
< l : c : l' >
```

where `l` is the list of numbers still to be sent by the sender, `c` is the current contents of the channel, and `l'` is the list of numbers already received by the receiver. The contents `c` can be either a natural number built up with `0` and the successor operator `s`, or the empty contents `mt`, or an acknowledgment `ack`. Lists of natural numbers are defined with the function symbols `nil`, `_;_`, and `_@_`, with `_;_` a list "cons" operator and `_@_` a list append operator. The equations $E^{\texttt{CHANNEL}}$ are labeled `[ap0]` and `[ap1]`. They define the append function in the usual way. The rules $R^{\texttt{CHANNEL}}$ specifying the system's transitions are labeled `[send]`, `[recv]`, and `[ack]`. Rule `[send]` puts the leftmost number of the sender's list in the channel if the channel is empty, rule `[recv]` appends the number in the channel to the receiver's list and sends back an `ack`, and rule `[ack]` consumes the `ack` message and clears the channel so that a new number can be sent.

Sort `Terminal` is the subsort of `State` determined by those states in which there are not numbers waiting to be sent through the channel and the chan-

nel is empty. The intention, of course, is to characterize the terminal or final states of the system, for which no more transitions are possible. First note that the only symbol not having the `ctor` declaration is the list append operator `_@_`. Therefore, $E^{\texttt{CHANNEL}}$-sufficient completeness is the claim that `_@_` is fully defined by the equations `[ap0]` and `[ap1]`. Analogously, the only symbol having the `ctor` declaration and not having the `metadata "rctor"` declaration is

```
op <_:_:_> : NilList EmptyChannel List
                    -> Terminal [ctor metadata "rctor"] .
```

Since only terms of sort `State` can be rewritten by the rules $R^{\texttt{CHANNEL}}$, this means that every state is expected to be rewritable with $R^{\texttt{CHANNEL}}$ to one of the form $< \texttt{nil} : \texttt{mt} : \texttt{l} >$.

Two key properties of `CHANNEL` are of particular interest:

1. *In-order reception*: every (ground) *terminal* state reachable from an *initial* state of the form $< \texttt{l} : \texttt{mt} : \texttt{nil} >$ preserves the order of messages, i.e., for `L` and `L'` variables of sort `List`:

$$\texttt{CHANNEL} \Vdash < \texttt{L} : \texttt{mt} : \texttt{nil} > \rightarrow < \texttt{nil} : \texttt{mt} : \texttt{L'} > \implies \texttt{L} = \texttt{L'}.$$

2. *Proper termination*: the protocol always terminates in a state of sort `Terminal`.

Observe that, if `CHANNEL` is strongly-normalizing and the constructor subsignature $\Upsilon^{\texttt{CHANNEL}}$ is a signature of terminal constructors, then (1) and (2) together ensure that the protocol always terminates with successful in-order communication. Note also that (1) cannot be checked by standard model-checking algorithms because the number of ground instances of `L` is countably infinite.

`CHANNEL` is executable (see Appendix A) and is sufficiently complete relative to its constructor signature pair, as shown below:

```
Maude> (scc-df CHANNEL .)
Checking sufficient completeness and deadlock freeness of CHANNEL...
Success: The equational subtheory of CHANNEL is sufficiently complete
  under the assumption that it is ground weakly-normalizing, ground confluent,
  and ground sort-decreasing.
Success: The rewrite theory CHANNEL is deadlock-free outside rctor-terms under
  the assumption that it is ground weakly-normalizing, ground sort-decreasing,
  and ground coherent.
```

This particularly implies that the reachability condition in (1) is satisfiable and hence the property is not trivially true. Two complementary proofs are

required for establishing (1), namely, a proof of the *existence* of a reachable terminal state preserving the order of messages for each initial state, and a proof of the *uniqueness* of such a terminal state. Property (2) follows directly from the strong-normalization of $\texttt{CHANNEL}_E$ (see Appendix A), plus the fact that $\Upsilon^{\texttt{CHANNEL}}$ is a signature of terminal constructors as just shown.

The existence claim is a logical consequence of the following inductive claim, for `L` and `L'` variables of sort `List`:

$$\texttt{CHANNEL} \vdash_{ind} < \texttt{L} : \texttt{mt} : \texttt{L'} > \rightarrow < \texttt{nil} : \texttt{mt} : \texttt{L}' \,@\, \texttt{L} > .$$

Using the sufficient completeness of `CHANNEL` relative to its constructor signature pair $(\Upsilon^{\texttt{CHANNEL}}, \Omega^{\texttt{CHANNEL}})$, the steps of the constructor-based inductive proof are a *base case* in which the property is proved for $l = \texttt{nil}$, and an *inductive case* in which the property is proved for $l = \texttt{n ; l}$ with the hypothesis that there is a proof for `n` and `l`, where `n` is a fresh constant of sort `Nat` and `l` is a fresh constant of sort `List`.

The soundness of the proof follows from the soundness of structural induction and from Theorem 5, where $C^{\perp}_{\texttt{CHANNEL,List}} = T_{\Upsilon^{\texttt{CHANNEL}},\texttt{List}}$. As a remark observe that the choice of induction variable for this proof does not increase its complexity, because $\Upsilon_{\texttt{CHANNEL,List}} = \Omega_{\texttt{CHANNEL,List}}$.

- *Base case.* Let $l = \texttt{nil}$:

  ```
    < nil :  mt :  L' >
  =   { L' @ nil = L' is an inductive consequence of E_CHANNEL }
    < nil :  mt :  L' @ nil >
  ```

- *Inductive case.* Assume the property holds for $l = \texttt{l}$, where `l` is a "fresh" constant of sort `List.` Let $l = \texttt{n ; l}$, with `n` a fresh constant of sort `Nat`:

  ```
    < (n ; l) :  mt :  L' >
  →1  { by [send] }
    < l :  n :  L' >
  →1  { by [recv] }
    < l :  ack :  L' @ (n ; nil) >
  →1  { by [ack] }
    < l :  mt :  L' @ (n ; nil) >
  →   { by induction hypothesis }
    < nil :  mt :  (L' @ (n ; nil)) @ l >
  ```

$=$ { by associativity of `@` is an inductive consequence of $\mathcal{E}_{\texttt{CHANNEL}}$ }
`< nil : mt : L' @ ((n ; nil) @ l) >`
$=$ { by `[ap01]` }
`< nil : mt : L' @ (n ; (nil @ l)) >`
$=$ { by `[ap02]` }
`< nil : mt : L' @ (n ; l) >`

The inductive claims about $\mathcal{E}_{\texttt{CHANNEL}}$ can be discharged automatically with the current version of Maude's Inductive Theorem Prover [52] by constructor-based equational induction over $E_{\texttt{CHANNEL}}$-constructors (see Appendix A).

The uniqueness proof uses a ground joinability argument about `CHANNEL`. As shown in Figure 3.2 for any simple generalized rewrite theory $\mathcal{R}$ that is admissible, if the rewrite rules $R \cup \overrightarrow{E}$ are ground sort-decreasing, ground confluent, and ground weakly-normalizing modulo B, then the ground confluence of $\mathcal{R}$ is a logical consequence of the ground confluence of $\mathcal{R}_{E \cup R}$. The key observation is that for rewrite proofs $\mathcal{R} \vdash t \to u$ and $\mathcal{R} \vdash t \to v$, there are analogous rewrite proofs $\mathcal{R}_{E \cup R} \vdash t \to u \downarrow_{E/B}$ and $\mathcal{R}_{E \cup R} \vdash t \to v \downarrow_{E/B}$. Since $\mathcal{R}_{E \cup R}$ is ground confluent, the (ground) $\mathcal{R}_{E \cup R}$-joinability witness for $u \downarrow_{E/B}$ and $v \downarrow_{E/B}$ is also a witness for the $\mathcal{R}$-joinability of u and v: membership to $E \cup B$-equivalence classes is invariant under sequent inference with the oriented equations in $\mathcal{R}_E$. In this way $\mathcal{R}$ inherits the ground confluence from $\mathcal{R}_{E \cup R}$. Note that in Figure 3.2 there is no need to mention $\mathcal{R}_E$ because $\mathcal{R}_{E \cup R}$ subsumes deduction with $\mathcal{R}_E$. Its mention is made explicit for better understanding of the proof. Also, note that $\mathcal{R}_E$ is trivially ground sort-decreasing, ground confluent, and ground operationally terminating modulo B because of the assumptions on $\mathcal{R}$.

`CHANNEL` is ground sort-decreasingness, ground confluence, and ground operationally terminating as automatically shown in Appendix A. This then entails the desired uniqueness proof of a reachable deadlock state preserving the order of messages for each initial state, and hence, property (1) holds. Property (2) follows directly from the strong-normalization of the rewrite theory $\texttt{CHANNEL}_{E^{\texttt{CHANNEL}} \cup R^{\texttt{CHANNEL}}}$.

Therefore, as desired, the `CHANNEL` protocol always terminates in a state of sort `Terminal` with successful in-order communication.

3.7 Related Work and Concluding Remarks

Sufficient completeness was first defined in Guttag's thesis [50]; this property is in general undecidable, even for unconditional equational specifica-

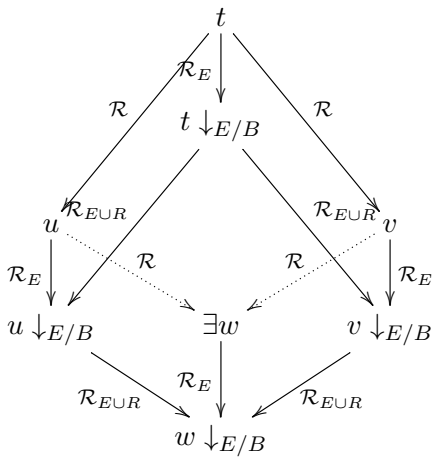

Figure 3.2: If $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$ is an admissible simple generalized rewrite theory and such that $\mathcal{R}_{E\cup R}$ is (ground) sort-decreasing, (ground) confluent and (ground) weakly-normalizing, then the (ground) confluence of $\mathcal{R}_{E\cup R}$ implies the (ground) confluence of $\mathcal{R}$. In this figure, e.g., $t \stackrel{\mathcal{R}}{\to} u$ is a short hand for $\mathcal{R} \vdash t \to u$.

tions [50, 51]. Sufficient completeness of equational specifications has been widely studied, see, e.g., [56, 78, 25, 59, 14, 16, 15, 64], but some of the proposed approaches are restricted to simple expressive formalisms, such as unsorted specifications or specifications without structural axioms, or assume strong properties such as termination and confluence. For a good review of the literature up to the 1980s and for important decidability/undecidability results see [62, 61]. A closely connected concept is *ground reducibility*, see, e.g., [83, 62, 26, 63, 28]. Tree automata methods have been used since the late 1980s for both sufficient completeness and ground reducibility, see, e.g., [26, 28, 54, 16], and Chapter 4 of [27] and references there. In the context of order-sorted and membership equational logic specifications, sufficient completeness has been studied in, e.g., [17, 53, 15], and for order-sorted specifications modulo axioms, including the context-sensitive case, in [54, 52].

The work presented here combines and generalizes two different research strands. On the one hand, it can be seen as a natural generalization from the case of equations E to that of both equations E and rules R, of the work in [54, 52] on (propositional) equational tree automata methods for checking sufficient completeness of left-linear equations modulo axioms for context-sensitive order-sorted specifications. On the other hand, it also generalizes the work by I. Gnaedig and H. Kirchner [44] on constructors for non-terminating rewrite systems in the following precise sense: the notion of sufficient completeness proposed in [44] exactly corresponds to that of $\mathcal{R}$-sufficient completeness in this work for the special case of a rewrite theory $\mathcal{R} = (\Sigma, \varnothing, R)$, where Σ has a single sort and there are no equations. The

treatment of the more general case of rewrite theories $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$ clarifies the important distinction between constructors for equations and constructors for rules, extends the ideas to the more general order-sorted case modulo axioms and with frozenness information, provides new tree automata *automated* techniques that complement the deductive narrowing-based techniques proposed in [44], and, to the best of the author's knowledge, investigates for the first time the use of $\mathcal{R}$-constructors (and E-constructors) for inductive proofs of ground reachability.

As usual, there is room for improvement. Since the goal in this work has been to obtain *automatic* techniques for checking the sufficient completeness of a rewrite theory, some restrictions have been imposed, such as treating only the order-sorted case (leaving out the case of membership equational theories), and also assuming that equations and rules are left-linear and unconditional. The notion of a sufficiently complete rewrite theory is equally meaningful and useful without these restrictions. Therefore, reasoning techniques that will allow such a property to be established for more general rewrite theories should be investigated, even if such techniques are no longer automatic. The related topic of constructor-based inductive techniques for ground reachability and ground joinability has only been sketched out; it deserves a much fuller development in future work, in which a detailed comparison with alternative approaches to proving such properties should also be given. Furthermore, these constructor-based induction techniques should be supported by tools such as, for example, an extension of the current Maude Inductive Theorem Prover (ITP).

CHAPTER 4

DEDUCTIVE PROOFS FOR SAFETY PROPERTIES

Safety properties of concurrent systems are among the most important properties to verify. They have received extensive attention in many different formal approaches, both algorithmic and deductive. Algorithmic approaches such as model checking are quite attractive because they are automatic. However, they cannot always be applied as a system can be infinite-state, so that no model checking algorithm which assumes a finite-state system can directly be used.

Even if an abstraction can be found to make the system finite-state, an additional difficulty may arise: although for each initial state the set of states reachable from it is finite, the set of initial states may still be *infinite*, so that model checking verification may not be possible. For example, a mutual exclusion protocol should be verified for an arbitrary number of clients in its initial state, even if the states have been abstracted away so that the set of states reachable from each initial state is always finite.

This chapter is part of a broader effort to develop *generic* methods to reason about safety properties of concurrent systems and more generally about any property specifiable in temporal logic. It advances such an effort by developing generic *deductive* methods and tools for proving two key safety properties, namely, stability and invariance, plus their combination by means strengthening techniques. The expression "generic" means that the verification methods and their associated tools are not tied to a specific programming language. By contrast, the UNITY logic is an elegant temporal logic inference system tailored for the verification of concurrent programs in the UNITY language [20], so that nontrivial changes would be required to apply such a logic to, say, threaded Java programs. Similarly, the deductive methods for verifying safety properties developed by Manna

and Pnueli in [68] are tailored to verify concurrent programs in the specific imperative language described in [68].

The advantage of generic verification methods and tools is that the costly tool development effort can be amortized across a much wider range of applications, whereas a language-specific verification tool can only be applied to systems programmed in that specific language. Of course, any such generic approach requires a *logical framework* general enough to encompass many different models and languages. In this case, the use of the rewriting logic framework [70] is justified by its ability to express very naturally many different models of concurrent computation and many concurrent languages. The generic framework and its tools can then be easily specialized to specific languages. This is exactly the approach taken in the rewriting logic semantics project [75], where the semantics of a wide variety of concurrent programming languages is defined in rewriting logic, and then Maude [23] and its LTL model checker can be used to verify programs in any of those languages.

The goal of this chapter is to extend rewriting-logic-based generic verification methods to support the *deductive* verification of concurrent systems, beginning with safety properties. In the rewriting logic framework, a concurrent system, such as, for example, a network protocol or an entire concurrent programming language such as Java, is specified as a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$, with (Σ, E) an equational theory specifying the system's *states* as elements of the initial algebra $\mathcal{T}_{\Sigma/E}$, and R a collection of (non-equational) rewrite rules specifying the system's *concurrent transitions*.

The generic approach presented here to safety property verification is both *transformational* and *reductionistic*. Safety properties are a special type of *inductive* properties. That is, they do not hold for just any model of the given rewrite theory $\mathcal{R}$, but for its *initial reachability model* $\mathcal{T}_{\mathcal{R}}$ (see Chapter 2). Concretely, for $\mathcal{R} = (\Sigma, E, R)$, this means that the states of such an initial model are precisely elements of the initial algebra $\mathcal{T}_{\Sigma/E}$, and that its one-step transitions are *provable* rewrite steps between such states by means of the rules R. Therefore, given any safety property φ, the interest here is in the model-theoretic satisfaction relation $\mathcal{T}_{\mathcal{R}} \models \varphi$, which is approximated deductively by means of an inductive inference relation $\mathcal{R} \Vdash \varphi$. This relation is proved *sound*, that is, $\mathcal{R} \Vdash \varphi$ always implies $\mathcal{T}_{\mathcal{R}} \models \varphi$.

This approach is *transformational* in the sense that the rules of inference transform pairs of the form $\mathcal{R} \Vdash \varphi$ into other such pairs $\mathcal{R}' \Vdash \varphi'$. It is also *reductionistic* in the sense that: (i) all safety formulas in temporal logic eventually disappear and are replaced by *purely equational formulas*, and (ii)

the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is eventually replaced by its underlying *equational theory* (Σ, E). That is, in the end *all formal reasoning about safety properties is reduced to inductive reasoning about equational properties* in the underlying equational theory (Σ, E). This allows these generic safety verification methods to take advantage of the existing wealth of equational reasoning techniques and tools already available.

The Maude Invariant Analyzer (InvA) tool supporting the transformational inference system presented in this chapter, makes systematic use of *narrowing modulo axioms* with the equations defining state predicates, specialized in this chapter to ground stability and invariance analysis to greatly simplify the equational proof obligations to which all proofs of safety formulas are ultimately reduced. It also takes full advantage of other heuristics for discharging automatically many proof obligations, all explained in what follows.

The main contributions of this chapter can be summarized as follows:

- Proof of the inductive soundness of a transformational inference system to prove stability and invariance properties about the initial reachability model $\mathcal{T}_{\mathcal{R}}$ of a topmost rewrite theory $\mathcal{R}$, as well as the soundness of additional inference rules supporting the *strengthening* of invariants.

- Systematic use of *narrowing modulo axioms* with the equations defining state predicates, specialized to ground stability and invariance analysis, to greatly simplify the equational proof obligations to which all proofs of safety formulas are ultimately reduced.

- Implementation of the above inference system in the InvA tool, which provides a substantial degree of automation and can automatically discharge many proof obligations without user intervention.

This chapter is organized as follows. A temporal semantics in the form of a Kripke structure is associated to a rewrite theory's initial reachability model in Section 4.1. An inference system for the deductive analysis of ground stability and ground invariance, including narrowing-based inference rules, is introduced and proved sound in Section 4.2. Section 4.3 discusses new inference rules that can be used to strengthen ground invariants. The Invariant Analyzer tool (InvA) is presented in Section 4.4, including a description of the main commands available to the user and the strategies it uses for discharging proof obligations. Related work and some final remarks can be found in Section 4.5. Case studies for these methods and the InvA tool are presented in chapters 5 and 6.

4.1 Temporal Semantics of $\mathcal{T}_\mathcal{R}$

The models of temporal logic are Kripke structures. A Kripke structure can be associated with the initial reachability model $\mathcal{T}_\mathcal{R}$ of a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$ by making explicit the intended set of *states* in $\mathcal{T}_\mathcal{R}$ and the relevant *state predicates* to the verification task.

In general, the state predicates need not be part of the system specification and therefore they need not be specified in $\mathcal{R}$. They are typically part of the *property specification*. This is mainly because state predicates need not be related to the operational semantics of $\mathcal{R}$, as they are just certain predicates about the states of $\mathcal{T}_\mathcal{R}$ that are need to specify some of its properties.

Therefore, after choosing a top sort in Σ, say $\mathfrak{s}$, for the set of states, the set of state predicates Π for $\mathcal{R}$ can be defined in an equational theory $\mathcal{E}_\Pi = (\Sigma_\Pi, E \cup E_\Pi)$. Signature Σ_Π contains Σ and a sort *Bool* with constant symbols $\top$ and $\bot$ of sort *Bool*, predicate symbols

$$p : \mathfrak{s} \;\; s_1 \;\; \cdots \;\; s_n \longrightarrow [Bool]$$

for each each predicate $p \in \Pi$ parametric on the tuple of sorts $s_1, \ldots, s_n$ in Σ, abbreviated $p \in \Pi_{s_1 \ldots s_n}$, and optionally some auxiliary function symbols. Equations E_Π define the predicate symbols in Σ_Π and the auxiliary function symbols, if any. The theory $\mathcal{E}_\Pi$ protects (Σ, E) and also the equational theory specifying *Bool* (see Chapter 2 for details).

Given a state predicate $p_{s_1,\ldots,s_n} \in \Pi$ and ground terms $t_i \in T_{\Sigma,s_i}$, with $1 \leq i \leq n$, $\mathcal{E}_\Pi$ then defines the semantics of p at state $t \in T_{\Sigma,\mathfrak{s}}$ in $\mathcal{T}_\mathcal{R}$ as follows:

$$\mathcal{T}_\mathcal{R}, t \models p(t_1, \ldots, t_n) \qquad \Longleftrightarrow \qquad \mathcal{E}_\Pi \vdash p(t, t_1, \ldots, t_n) = \top. \tag{4.1}$$

This defines the Kripke structure:

$$\mathcal{K}^\Pi_\mathcal{R} = (T_{\Sigma/E,\mathfrak{s}}, \rightarrow_\mathcal{R}, L_\Pi), \tag{4.2}$$

with labeling function L_Π defined for each $t \in T_{\Sigma,\mathfrak{s}}$ and $t_i \in T_{\Sigma,s_i}$, with $1 \leq i \leq n$, by:

$$p(t_1, \ldots, t_n) \in L_\Pi(t) \qquad \Longleftrightarrow \qquad \mathcal{E}_\Pi \vdash p(t, t_1, \ldots, t_n) = \top. \tag{4.3}$$

In this way, all of LTL can be interpreted in $\mathcal{K}^\Pi_\mathcal{R}$ in the standard way [22], including also the first-order version of LTL.

Note that only the positive case is needed to define p's semantics. The

reason why p has codomain $[Bool]$ instead of $Bool$, is to allow partial definitions of p with equations that only define the *positive* case by equations $p(t, t_1, \ldots, t_n) = \top$ **if** γ, and either leave the *negative* case implicit or may only define some negative cases with equations $p(t', t'_1, \ldots, t'_n) = \bot$ **if** γ' without necessarily covering all the cases, i.e., without p's definition having to be sufficiently complete. This possibly partial specification of predicates (yet, with *full* specification in the *positive* case) can be very convenient, since the full definition of the negative cases can sometimes be quite involved. However, the sort $Bool$ is *protected*: only when a term $p(t)$ can be proved equal to either $\top$ or $\bot$ can the term $p(t)$ have sort $Bool$. Nevertheless, for proving purposes it is often useful to define some negative cases for which a state predicate p does not hold, since this helps in discarding proof obligations in the form of an implication whose antecedent is false.

It is also important to note that a state predicate $p \in \Pi$ can act as a *definitional extension* of a Boolean combination of other state predicates $\{p_1, \ldots, p_n\}$ in Σ_Π, so that the choice of focusing on atomic state predicates is mainly to simplify the exposition but does not limit the general applicability of the results that follow. In a rewriting logic language implementation such as Maude [23], definitional extensions can be conveniently obtained by having $\mathcal{E}_\Pi$ protecting Maude's predefined functional module `BOOL-OPS`, which declares constants $\top$ and $\bot$ of sort $Bool$ along with Boolean function symbols such as conjunction, disjunction, negation, etc.

4.2 Ground Safety Properties

The development in section assumes a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$, with order-sorted signature $\Sigma = (S, \leq, F)$ and with top sort $\mathfrak{s} \in S$ for the set of states, and a set Π of state predicates for $\mathcal{R}$ equationally defined in $\mathcal{E}_\Pi = (\Sigma_\Pi, E \cup E_\Pi)$. Moreover, expressions of the form $\vec{t}$ are used to abbreviate lists of terms $t_1, \ldots, t_n$ in $T_\Sigma(X)$, and thus simplify notation.

4.2.1 Ground Stability

The concept of ground stability for $\mathcal{R}$ is intimately related with the notion of the set of states $t \in T_{\Sigma/E,\mathfrak{s}}$ of $\mathcal{T}_\mathcal{R}$ that satisfy a state predicate p being closed under $\rightarrow_\mathcal{R}$. More precisely, for $p \in \Pi_{s_1,\ldots,s_n}$ and $\vec{x} = x_{s_1}, \ldots, x_{s_n}$ variables

in X, the property p being ground stable for $\mathcal{R}$ is the safety property:

$$\mathcal{K}^{\Pi}_{\mathcal{R}} \models p(\vec{x}) \Rightarrow \Box p(\vec{x}), \tag{4.4}$$

meaning that for any ground substitution $\sigma : X \longrightarrow T_\Sigma$ if $p(\vec{x}\sigma)$ holds in a state $t \in T_{\Sigma,\mathfrak{s}}$, then $p(\vec{x}\sigma)$ holds in any state $u \in T_{\Sigma,\mathfrak{s}}$ that is reachable from t.

Definition 13 (Ground Stability)**.** *Let $p \in \Pi_{s_1,\ldots,s_n}$ and let $\vec{x} = x_{s_1}, \ldots, x_{s_n}$ be variables in X. Then:*

- *$\mathcal{R}$ is* ground p-stable under *rules $R_0 \subseteq R$ if and only if for all $t, u \in T_{\Sigma,\mathfrak{s}}$ and ground substitution $\sigma : X \longrightarrow T_\Sigma$ the following implication holds:*

$$\mathcal{E}_\Pi \Vdash p(t, \vec{x}\sigma) = \top \wedge (\Sigma, E, R_0) \Vdash t \rightarrow^* u \implies \mathcal{E}_\Pi \vdash p(u, \vec{x}\sigma) = \top. \tag{4.5}$$

- *$\mathcal{R}$ is* ground p-stable*, written $\mathcal{R} \Vdash p(\vec{x}) \Rightarrow \Box p(\vec{x})$, if and only if $\mathcal{R}$ is ground p-stable under R.*

Note that the deduction relation '$\Vdash$' in Definition 13 shares the same meaning as the notation in Section 3.5 introduced for constructor-based reachability analysis. This relation refers to deduction, both at the equational and rewrite theory levels, in the initial models, that is, to deduction where variables range over ground terms only.

The reachability condition in the Definition 13 can be reduced to a simpler 1-step rewrite condition, resulting in an equivalent notion of ground stability that avoids arbitrary depth proof search. In the notation of Linear Time Temporal Logic (LTL), this is captured by the inference rule ST in Figure 4.1. This rule greatly simplifies the LTL reasoning about the p-stability of the Kripke structure $\mathcal{K}^{\Pi}_{\mathcal{R}} = (T_{\Sigma/E,\mathfrak{s}}, \rightarrow_{\mathcal{R}}, L_\Pi)$ associated to $\mathcal{R}$ and Π. Symbol "$\bigcirc$" corresponds to the next operator in LTL and symbol "$\Rightarrow$" to strong implication in LTL (see [67] for details). So, for $\mathcal{K}^{\Pi}_{\mathcal{R}} \models p(\vec{x}) \Rightarrow \Box p(\vec{x})$ to hold, it is enough to show that $\mathcal{K}^{\Pi}_{\mathcal{R}} \models p(\vec{x}) \Rightarrow \bigcirc p(\vec{x})$ holds.

Lemma 1 proves that the inference rule ST is not only sound but also complete.

Lemma 1. *Inference rule* G-ST *in Figure 4.1 is sound and complete.*

Proof. Let $t, u \in T_{\Sigma,\mathfrak{s}}$, $\mathcal{R}_0 = (\Sigma, E, R_0)$, $\sigma : X \longrightarrow T_\Sigma$, and $\vec{x} = x_1, \ldots, x_n \in X$. ($\implies$) By assumption, $\mathcal{E}_\Pi \vdash p(t, \vec{x}\sigma) = \top$ and $\mathcal{R}_0 \vdash t \rightarrow u$. The

$$\frac{\mathcal{R} \Vdash p(\vec{x}) \Rightarrow \bigcirc p(\vec{x})}{\mathcal{R} \Vdash p(\vec{x}) \Rightarrow \Box p(\vec{x})}\ \textsc{St}$$

$$\frac{\bigwedge_{\substack{(l \to r \textbf{ if } \gamma) \in R \\ (\theta, w, \gamma', \vec{v}) \in \Theta_{(l,r,\gamma)}}} \mathcal{E}_\Pi \Vdash p(r\theta, \vec{v}) = \top \textbf{ if } \gamma\theta \wedge \gamma'\theta \wedge w\theta = \top}{\mathcal{R} \Vdash p(\vec{x}) \Rightarrow \bigcirc p(\vec{x})}\ \textsc{Nr1}$$

Figure 4.1: Ground p-stability for $\mathcal{R} = (\Sigma, E, R)$, with Θ defined as in Theorem 7.

latter fact implies that $\mathcal{R}_0 \vdash t \to^* u$. Then, from the hypothesis, it follows that $p(u, \vec{x}\sigma) = \top$. ($\Longleftarrow$) By induction on the number m of rewrite steps proving $\mathcal{R}_0 \vdash t \to^m u$. If $m = 0$, then $\mathcal{R}_0 \vdash t = u$ and by definition $t =_E u$. Since $\mathcal{E}_\Pi \vdash p(t, \vec{x}\sigma) = \top$, it must be the case that $\mathcal{E}_\Pi \vdash p(u, \vec{x}\sigma) = \top$. If $\mathcal{R}_0 \vdash t \to^{m+1} u$, then there is $u_0 \in T_{\Sigma,\mathfrak{s}}$ such that $\mathcal{R}_0 \vdash t \to u_0 \wedge u_0 \to^m u$. If $\mathcal{E}_\Pi \vdash p(t, \vec{x}\sigma) = \top$, then $\mathcal{E}_\Pi \vdash p(u_0, \vec{x}\sigma) = \top$. Moreover, from assumption and the induction hypothesis $\mathcal{R}_0 \vdash u_0 \to^m u$, it follows that $\mathcal{E}_\Pi \vdash p(u, \vec{x}\sigma) = \top$. □

The next question to ask is how to reduce the verification of the simpler condition $p \Rightarrow \bigcirc p$ to inductive equational reasoning. For this purpose, the idea of (one-step) *narrowing with equations modulo axioms* [58], a sound and complete method for ground stability analysis, is used to reduce the inductive reachability problem of p-stability for $\mathcal{T}_\mathcal{R}$ to equational inductive properties of $\mathcal{T}_{(\Sigma,E)}$.

Under admissibility assumptions, the equations in $\mathcal{R}$ are a disjoint union $E \uplus B$ of ground sort-decreasing, ground operationally terminating, ground confluent, and ground coherent (w.r.t. the rules R) equations E modulo structural axioms B such as associativity, commutativity, and identity. For a combination of free and associative and/or commutative and/or identity axioms, except for symbols f that are associative but not commutative, a finitary B-unification algorithm exists. Instead, in general there is no finitary $E \cup B$-unification algorithm. However, for $\Omega \subseteq \Sigma$ a signature of free equational constructors modulo B (see Chapter 3) and an Ω-equality $t = u$, the ground instances of $CSU_B(t = u)$ exactly characterize as its ground instances the set $GU_{E \cup B}(t = u)$.

Lemma 2. *Let $\mathcal{E} = (\Sigma, E \cup B)$ be an admissible order-sorted equational theory with finitary B-unification algorithm and with $\Omega \subseteq \Sigma$ a signature of free equational constructors modulo B. Then, for any $t, u \in T_\Omega$, the*

following equivalence holds:

$$\alpha \in GU_{E\cup B}(t = u) \\ \iff (\exists \theta \in CSU_B(t = u))(\exists \sigma : X \longrightarrow T_\Omega)\, \theta\sigma =_{E\cup B} \alpha. \tag{4.6}$$

Proof. Let $t, u \in T_\Omega(X)_s$ for some sort s in Σ. ($\implies$) Assume $\alpha \in GU_{E\cup B}(t = u)$, i.e., $\alpha : X \longrightarrow T_\Sigma$ is such that $t\alpha =_{E\cup B} u\alpha$. Since Ω is a subsignature of E-free constructors, there is $\beta : X \longrightarrow T_\Omega$ satisfying $\alpha =_{E\cup B} \beta$ and such that $\alpha(x) \downarrow_{\Sigma,E/B=B} \beta(x)$, for $x \in X$. Consequently $t\beta =_B u\beta$. Hence, there is $\theta \in CSU_B(t = u)$ and $\sigma : X \longrightarrow T_\Omega$ such that $\theta\sigma =_B \beta =_{E\cup B} \alpha$. ($\impliedby$) Suppose $\theta \in CSU_B(t = u)$ and let $\sigma : X \longrightarrow T_\Omega$ be a ground substitution. Then, it follows that $t\theta\sigma =_B u\theta\sigma$ and, a fortiori, $t\theta\sigma =_{E\cup B} u\theta\sigma$. Hence, $\theta\sigma \in GU_{E\cup B}(t = u)$ as desired. □

In order to show the ground p-stability of $\mathcal{R}$ the approach is to prove, for each rule $l \to r$ **if** $\gamma \in R$, that if for a ground instance $l\sigma$ of l the predicate p and the condition $\gamma\sigma$ hold, then $p(\vec{x}\sigma)$ must hold in state $r\sigma$. Since by assumption $l \in T_{\Omega,\mathfrak{s}}(X)$, the key observation here is that, if all left hand-sides $p(v, \vec{v})$ of equations $p(v, \vec{v}) = w$ **if** $\gamma' \in E_\Pi$ defining the state predicate $p \in \Pi$ are Ω-*patterns* in the state parameter v (i.e., $v \in T_{\Omega(X),\mathfrak{s}}$), then $CSU_B(l = v)$ can be computed to obtain substitutions θ which, by Lemma 2, exactly characterize any ground $E \cup B$-unifier in $GU_{E\cup B}(l = v)$. Each substitution $\theta \in CSU_B(l = v)$ is such that $p(l\theta, \vec{v}) = \top$, or at least $p(l\theta, \vec{v})$ *could* be equal to $\top$. Hence, all that is left is the task of inductively proving $p(r\theta, \vec{v}) = \top$ under the assumptions $\gamma\theta$, $\gamma'\theta$, and $w\theta = \top$. In this way, the inductive reachability problem of p-stability for $\mathcal{T}_\mathcal{R}$ is recast as the problem of proving simpler equational inductive properties of $\mathcal{T}_{\Sigma/E\cup B}$: $\mathcal{T}_\mathcal{R}$ is ground p-stable if and only if $\mathcal{T}_{\Sigma/E\cup B}$ satisfies these inductive properties, as stated by the narrowing inference rule NR1 in Figure 4.1.

Theorem 7 proves soundness and completeness of the narrowing inference rule NR1 in Figure 4.1.

Theorem 7. *Let $\mathcal{R} = (\Sigma, E \cup B, R)$ be admissible with signature $\Omega \subseteq \Sigma$ of (equational) free constructors modulo B. Let $p \in \Pi_{s_1,\ldots,s_n}$ and $l \to r$ **if** $\gamma \in R$. Without loss of generality, assume that the equations $E^p_\Pi \subseteq E_\Pi$ defining p are all conditional, have no variables in common with the rewrite rule, and have Ω-patterns as left-hand sides in the state parameter. If*

$$\Theta_{(l,r,\gamma)} = \bigcup_{(p(v,\vec{v})=w \text{ if } \gamma')\in E^p_\Pi} \{(\theta, w, \gamma', \vec{v}) \mid \theta \in CSU_B(v = l)\},$$

then inference rule NR-1 *in Figure 4.1 is sound and complete.*

Proof. In the following proof, some equations and rules explicitly mention the variables occurring in them, and also the domain of some substitutions is restricted to subsets of X. Let $R_0 = \{(\forall Y)\ l \to r \text{ if } C\}$ and $\mathcal{R}_0 = (\Sigma, E \cup B, R_0)$. In order to simplify the proof, and without loss of generality, assume that $p \in \Pi$ is such that it has no parameters.

$\mathcal{R}$ is ground p-stable under R_0
$\iff$ { by definition of ground p-stability and Lemma 1 }
$(\forall t, u \in T_{\Sigma,\mathfrak{s}})$
$\quad \mathcal{E}_\Pi \vdash p(t) = \top \wedge \mathcal{R}_0 \vdash t \to u \Longrightarrow \mathcal{E}_\Pi \vdash p(u) = \top$
$\iff$ { by definition of rewriting and by $\mathcal{E}_{\mathcal{R}_0} = \mathcal{E}_\mathcal{R}$ }
$(\forall \alpha : Y \longrightarrow T_\Sigma)$
$\quad \mathcal{E}_\Pi \vdash p(l\alpha) = \top \wedge \mathcal{E}_\mathcal{R} \vdash \gamma\alpha \implies \mathcal{E}_\Pi \vdash p(r\alpha) = \top$
$\iff$ { by $\mathcal{E}_\Pi$ protecting $\mathcal{E}_\mathcal{R}$ and $\gamma\alpha$ a ground Σ-formula }
$(\forall \alpha : Y \longrightarrow T_\Sigma)$
$\quad \mathcal{E}_\Pi \vdash p(r\alpha) = \top$ **if** $p(l\alpha) = \top \wedge \gamma\alpha$
$\iff$ { by $\mathcal{E}_\Pi$ ground confluent, with $Z_v = vars(v)$ }
$(\forall \alpha : Y \longrightarrow T_\Sigma)(\forall (p(v) = w \textbf{ if } \gamma) \in E^p_\Pi)(\forall \beta : Z_v \longrightarrow T_\Sigma)$
$\quad \mathcal{E}_\Pi \vdash p(r\alpha) = \top$ **if** $l\alpha = v\beta \wedge \gamma'\beta \wedge w\beta = \top \wedge \gamma\alpha$
$\iff$ { by assumption $Y \cap Z_v = \varnothing$, with $\eta = \alpha \cup \beta$ and $X_v = Y \cup Z_v$ }
$(\forall (p(v) = w \textbf{ if } \gamma') \in E^p_\Pi)(\forall \eta : X_v \longrightarrow T_\Sigma)$
$\quad \mathcal{E}_\Pi \vdash p(r\eta) = \top$ **if** $l\eta = v\eta \wedge \gamma'\eta \wedge w\eta = \top \wedge \gamma\eta$
$\iff$ { by Lemma 2: $l, v \in T_\Omega(X)_\mathfrak{s}$ and $\mathcal{E}_\Pi$ protecting $\mathcal{E}_\mathcal{R}$ }
$(\forall (p(v) = w \textbf{ if } \gamma') \in E^p_\Pi)(\forall \theta \in CSU_B(l = v)B)(\forall \sigma : ran(\theta) \longrightarrow T_\Sigma)$
$\quad \mathcal{E}_\Pi \vdash p(r\theta\sigma) = \top$ **if** $\gamma'\theta\sigma \wedge w\theta\sigma = \top \wedge \gamma\theta\sigma$
$\iff$ { by definition of $\Vdash$ }
$(\forall (p(v) = w \textbf{ if } \gamma') \in E^p_\Pi)(\forall \theta \in CSU_B(l = v))$
$\quad \mathcal{E}_\Pi \Vdash p(r\theta) = \top$ **if** $\gamma\theta \wedge \gamma'\theta \wedge w\theta = \top$
$\iff$ { by definition of $\Theta_{(l,r,\gamma)}$ }
$(\forall (\theta, w, \gamma') \in \Theta_{(l,r,\gamma)})$
$\quad \mathcal{E}_\Pi \Vdash p(r\theta) = \top$ **if** $\gamma\theta \wedge \gamma'\theta \wedge w\theta = \top$

□

Observe that obtaining a complete set of unifiers in the definition of $\Theta_{(l,r,\gamma)}$ in Theorem 7 only involves Σ-terms and not Σ_Π-terms. This is useful in practice because the generation of proof obligations from $\Theta_{(l,r,\gamma)}$ does not depend on the state predicates defined in $\mathcal{E}_\Pi$ and therefore is not affected by their equational definitions, no matter how involved these definitions may be.

Also observe that, since the complete set of B-unifiers is finite, the set $\Theta_{(l,r,\gamma)}$ is also finite for each $l \to r$ **if** $\gamma \in R$. Therefore, the set of proof obligations is finite because of the finiteness assumptions on E and R. As a final remark, observe that when w is $\bot$ in an equation $p(v, \vec{v}) = w$ **if** $\gamma' \in E^p_\Pi$, each proof obligation $p(r\theta, \vec{v}) = \top$ **if** $\gamma\theta \wedge \gamma'\theta \wedge w\theta = \top$ can be *soundly* ignored, because $w\theta = \bot\theta = \bot \neq \top$ since $\mathcal{E}_\Pi$ protects the sort *Bool*.

4.2.2 Ground Invariance

Invariants are among the most important safety properties. Given a set of initial states characterized by $I \in \Pi_{s'_1,\ldots,s'_m}$, a state predicate $p \in \Pi_{s_1,\ldots,s_n}$ being *ground invariant* for $\mathcal{R}$ from the set of initial states I is the safety property

$$\mathcal{K}^\Pi_\mathcal{R} \models I(x_{s'_1}, \ldots, x_{s'_m}) \Rightarrow \Box p(x_{s_1}, \ldots, x_{s_n}), \tag{4.7}$$

where the s_i and s'_j (resp., the x_{s_i} and x'_{s_j}) need not be different, meaning that for any ground substitution $\sigma : X \longrightarrow T_\Sigma$, if $I(x_{s'_1}, \ldots, x_{s'_m})\sigma$ holds in a state $t \in T_{\Sigma,\mathfrak{s}}$, then $p(x_{s_1}, \ldots, x_{s_n})\sigma$ holds in any state $u \in T_{\Sigma,\mathfrak{s}}$ reachable from t. In other words, the invariant p holds for all states reachable from I. Since the set of initial states is defined in $\mathcal{E}_\Pi$ as a state predicate $I \in \Pi$, an equational definition of I can of course capture an infinite set of initial states, even in the case when I has no parameters.

In what follows, it is assumed that the states predicates in Π are parametric on the same tuple of sorts, say $s_1, \ldots, s_n$. This does help in keeping the syntax cleaner and the definitions simpler, without affecting the generality of the approach.

Definition 14 (Ground Invariance)**.** *Let $p, I \in \Pi_{s_1,\ldots,s_n}$ be state predicates and $\vec{x} = x_{s_1}, \ldots, x_{s_n} \in X$. Then:*

- *$\mathcal{R}$ is* ground *p*-invariant from *I* under *rules $R_0 \subseteq R$ if and only if for all $t, u \in T_{\Sigma,\mathfrak{s}}$ and ground substitution $\sigma : X \longrightarrow T_\Sigma$ the following implication holds:*

$$\mathcal{E}_\Pi \vdash I(t, \vec{x}\sigma) = \top \wedge (\Sigma, E, R_0) \vdash t \to^* u \implies \mathcal{E}_\Pi \vdash p(u, \vec{x}\sigma) = \top. \tag{4.8}$$

- *$\mathcal{R}$ is* ground *p*-invariant from *I, written $\mathcal{R} \Vdash I(\vec{x}) \Rightarrow \Box p(\vec{x})$, if and only if $\mathcal{R}$ is ground p-invariant from I under R.*

Ground p-invariance for $\mathcal{R}$ is intimately related to its ground p-stability in the sense that if every initial state defined by a predicate I satisfies p and $\mathcal{R}$ is p-stable, then $\mathcal{R}$ is p-invariant from I. Of course, the converse does not necessarily hold, because even if $\mathcal{R}$ is ground p-invariant from I, the set of states of $\mathcal{T}_{\mathcal{R}}$ satisfying p need not be closed under $\rightarrow_{\mathcal{R}}$. The key observation is that in $\mathcal{T}_{\mathcal{R}}$, when every initial state defined by I satisfies p, the set of states satisfying p characterizes an over-approximation of the set of reachable states from the set of initial states specified by I.

In LTL terms, Lemma 3 justifies the soundness of the inference rule Inv in Figure 4.2 for proving that p is an invariant from I in the Kripke structure $\mathcal{K}_{\mathcal{R}}^{\Pi}$.

Lemma 3. *Inference rule* Inv *in Figure 4.2 is sound.*

Proof. Let $t, u \in T_{\Sigma,\mathfrak{s}}$, $\mathcal{R}_0 = (\Sigma, E, R_0)$, and $\sigma : X \longrightarrow T_\Sigma$ be a ground substitution. Assume (i) $\mathcal{E}_\Pi \vdash I(t, \vec{x}\sigma) = \top$ and (ii) $\mathcal{R}_0 \vdash t \rightarrow^* u$. From (1) and (i) it follows that $\mathcal{E}_\Pi \vdash p(t, \vec{x}\sigma) = \top$, and then from (2) and (ii), and the latter claim, follows that $\mathcal{E}_\Pi \Vdash p(u, \vec{x}\sigma) = \top$, as desired. □

For any state predicates $p, q \in \Pi_{s_1,\ldots,s_n}$ and $\vec{x} = x_1, \ldots, x_n \in X$, let

$$q(\vec{x}) \Rightarrow p(\vec{x})$$

be a shorthand for $p(x_{\mathfrak{s}}, \vec{x}) = \top$ **if** $q(x_{\mathfrak{s}}, \vec{x})$, where the x_i are assumed different from $x_{\mathfrak{s}}$. Condition 1 in Lemma 3 states that every initial state specified by I must satisfy property p, abbreviated $\mathcal{R} \Vdash I(\vec{x}) \Rightarrow p(\vec{x})$. Observe that this condition does not depend on the dynamics of $\mathcal{T}_{\mathcal{R}}$, but only on its set of states $T_{\Sigma/E,\mathfrak{s}}$. The premises in inference rule Inv are used in the literature to cast the notion of *inductive invariant*, i.e., of a predicate holding in the set of initial states and being maintained true by every transition.

The only remaining question is how to prove $I(\vec{x}) \Rightarrow p(\vec{x})$. This question is answered by Theorem 8, which gives a necessary and sufficient condition for proving statements of this form. Theorem 8 justifies the soundness of the the inference rule C⇒ in Figure 4.2.

Theorem 8. *Inference rule* C⇒ *in Figure 4.2 is sound.*

Proof. ($\Longrightarrow$) Let $q(v, v_1, \ldots, v_n) = w$ **if** $\gamma' \in E_\Pi^q$ and assume $\mathcal{E}_\Pi \vdash \gamma'\sigma \wedge w\sigma = \top$ for some ground substitution $\sigma : X \longrightarrow T_\Sigma$. The goal is to show $\mathcal{E}_\Pi \vdash p(v, v_1, \ldots, v_n)\sigma = \top$. Note that $\mathcal{E}_\Pi \vdash \gamma\sigma \wedge w\sigma = \top$ implies $\mathcal{E}_\Pi \vdash q(v, v_1, \ldots, v_n)\sigma = \top$ witnessed by equation $q(v, v_1, \ldots, v_n) = w$ **if** γ', and therefore $\mathcal{E}_\Pi \vdash p(v, v_1, \ldots, v_n)\sigma = \top$ by hypothesis. ($\Longleftarrow$) Let $t \in T_{\Sigma,\mathfrak{s}}$

$$\frac{\mathcal{R} \Vdash I(\vec{x}) \Rightarrow p(\vec{x}) \qquad \mathcal{R} \Vdash p(\vec{x}) \Rightarrow \Box p(\vec{x})}{\mathcal{R} \Vdash I(\vec{x}) \Rightarrow \Box p(\vec{x})}\ \textsc{Inv}$$

$$\frac{\mathcal{E}_\Pi \Vdash \bigwedge_{(q(v,\vec{v})=w \text{ if } \gamma') \in E^q_\Pi} p(v,\vec{v}) = \top \text{ if } \gamma' \wedge w = \top}{\mathcal{R} \Vdash q(\vec{x}) \Rightarrow p(\vec{x})}\ \text{C}\Rightarrow$$

Figure 4.2: $\mathcal{R}$ ground p-invariance from I.

and $\sigma : X \longrightarrow T_\Sigma$. Assume $\mathcal{E}_\Pi \vdash q(t, \vec{x}\sigma) = \top$. The goal is to prove $\mathcal{E}_\Pi \vdash q(t, \vec{x}\sigma)$. Then, there is $q(v, v_1, \ldots, v_n) = w$ **if** $\gamma' \in E^q_\Pi$ and ground substitution $\rho : X \longrightarrow T_\Sigma$ such that $\mathcal{E}_\Pi \vdash \gamma'\rho \wedge w\rho = \top$, $t =_E v\rho$, and $\rho(v_i) =_E \sigma(x_i)$, for $1 \leq i \leq n$. Then, it follows from the assumption that $\mathcal{E}_\Pi \vdash p(v, v_1, \ldots, v_n)\rho = \top$, that is, $\mathcal{E}_\Pi \vdash p(t, \vec{x})\sigma = \top$ since $\mathcal{E} \subseteq \mathcal{E}_\Pi$. $\Box$

4.3 Strengthenings for Ground Invariance

Strengthening of invariants is a key technique for verifying safety properties. This section presents two strengthening techniques for ground invariance and proves their correctness. In what follows, $\mathcal{R} = (\Sigma, E, R)$, $\Sigma = (S, \leq, F)$, $\mathfrak{s} \in S$, and $\Pi = (\Sigma_\Pi, E \cup E_\Pi)$ are as assumed in Section 4.2. It is also assumed that the states predicates in Π are parametric on the same tuple of sorts, say $s_1, \ldots, s_n$, $x_\mathfrak{s} \in X_\mathfrak{s}$, and $\vec{x} = x_{s_1}, \ldots, x_{s_n}$ are variables in X different from $x_\mathfrak{s}$.

For state predicates $p, I \in \Pi_{s_1,\ldots,s_n}$, a *strengthening* for the ground p-invariance from I of a topmost rewrite theory $\mathcal{R}$ is given by a state predicate $q \in \Pi_{s_1,\ldots,s_n}$ such that $\mathcal{R}$ is ground q-invariant from I and, moreover, q can be used to prove $\mathcal{R} \Vdash I(\vec{x}) \Rightarrow \Box p(\vec{x})$. Traditionally, state predicate q is the result of a gradual refinement of a too-weakly defined p for which $\mathcal{R}$ being ground p-invariant cannot be proved directly by means of inference rule Inv in Figure 4.2.

Recall that inference rule Inv says that if $I(\vec{x}) \Rightarrow p(\vec{x})$ and $\mathcal{R}$ is ground p-stable, then $\mathcal{R}$ is ground p-invariant from I. The first key observation for an strengthening technique is the following: $\mathcal{R}$ may be ground p-invariant from I and yet not be ground p-stable. For ground p-invariance from I the only states from which p need not be falsified are precisely those states reachable from a state in I. The idea is then to strengthen p as follows: if $\mathcal{R}$ is ground q-invariant from I and every state satisfying q also satisfies p (i.e.,

$q(\vec{x}) \Rightarrow p(\vec{x})$), then clearly $\mathcal{R}$ is ground p-invariant from I. This is because any state in $\mathcal{T}_{\mathcal{R}}$ reachable from I satisfies p.

Theorem 9 states that for proving $\mathcal{R} \Vdash I(\vec{x}) \Rightarrow \Box p(\vec{x})$ by assuming $\mathcal{R} \Vdash J(\vec{x}) \Rightarrow \Box q(\vec{x})$, it is sufficient to equationally check the inductive validity of $q(\vec{x}) \Rightarrow p(\vec{x})$ and $I(\vec{x}) \Rightarrow J(\vec{x})$. In LTL terms, Theorem 9 proves the soundness of inference rule STR1 in Figure 4.3.

Theorem 9. *Inference rule* STR1 *in Figure 4.3 is sound.*

Proof. Let $t, u \in T_{\Sigma,\mathfrak{s}}$ and $\sigma : X \longrightarrow T_{\Sigma}$, assume that (i) $\mathcal{E}_{\Pi} \vdash I(t, \vec{x}\sigma) = \top$ and (ii) $\mathcal{R} \vdash t \rightarrow^* u$. The goal is to prove that $\mathcal{E}_{\Pi} \vdash p(u, \vec{x}\sigma) = \top$. From (1) and (i) follows $\mathcal{E}_{\Pi} \vdash J(t, \vec{x}\sigma) = \top$. This, (2), and (ii) imply $\mathcal{E}_{\Pi} \vdash q(u, \vec{x}\sigma) = \top$, which together with (3) imply $\mathcal{E}_{\Pi} \vdash p(u, \vec{x}\sigma) = \top$, as desired. □

The second strengthening technique is inspired by the following inductive observation. If $\mathcal{R}$ is ground q-invariant from I, then any state reachable from an initial state in I satisfies q. In particular, if t is a state from which a transition falsifies p's stability, then q may be used to to prune those spurious states by strengthening the condition of the stability proof obligation with the additional information of q and t.

Rule STR2 in Figure 4.3 formalizes this strengthening, which is proved sound in Theorem 10.

Theorem 10. *Inference rule* STR2 *in Figure 4.3 is sound.*

Proof. Let $t, u \in T_{\Sigma,\mathfrak{s}}$ and let $\sigma : X \longrightarrow T_{\Sigma}$ be a ground substitution. Assume that: (i) $I(\vec{x}\sigma)$ holds in state t and (ii) $t \rightarrow^* u$. The goal is to prove that $p(\vec{x}\sigma)$ holds in u. By induction on the number m of rewrite steps proving $\mathcal{R} \vdash t \rightarrow^m u$. If $m = 0$, then $\mathcal{R} \vdash t = u$ and since $t, u \in T_{\Sigma,\mathfrak{s}}$ and $\mathcal{E}_{\Pi}$ protects $\mathcal{E}$, it follows that $t =_E u$. Then, from (i) and the first premise in the inference rule, $\mathcal{E}_{\Pi} \vdash p(u, \vec{x}\sigma) = \top$. If $\mathcal{R} \vdash t \rightarrow^{m+1} u$, then there is a state $u_0 \in T_{\Sigma,\mathfrak{s}}$ satisfying $\mathcal{R} \vdash t \rightarrow^m u_0 \wedge u_0 \rightarrow u$. By the induction hypothesis $\mathcal{E}_{\Pi} \vdash p(u_0, \vec{x}\sigma) = \top$ holds. From the second premise, $\mathcal{E}_{\Pi} \vdash q(u_0, \vec{x}\sigma) = \top$ holds. These two facts and the third premise in the rule, imply the goal $\mathcal{E}_{\Pi} \vdash p(u, \vec{x}\sigma) = \top$. □

Theorem 11 proves the soundness of the narrowing inference rule NR2 in Figure 4.3.

Theorem 11. *Let* $\mathcal{R} = (\Sigma, E \cup B, R)$ *be admissible with signature* $\Omega \subseteq \Sigma$ *of free constructors modulo* B. *Let* $p, q \in \Pi_{s_1,\ldots,s_n}$ *and* $l \rightarrow r$ **if** $\gamma \in R$. *Without loss of generality, assume that the equations* $E^p_{\Pi} \subseteq E_{\Pi}$ *defining* p

$$\frac{\mathcal{R} \Vdash I(\vec{x}) \Rightarrow J(\vec{x}) \qquad \mathcal{R} \Vdash J(\vec{x}) \Rightarrow \Box q(\vec{x}) \qquad \mathcal{R} \Vdash q(\vec{x}) \Rightarrow p(\vec{x})}{\mathcal{R} \Vdash I \Rightarrow \Box p} \text{ STR1}$$

$$\frac{\mathcal{R} \Vdash I(\vec{x}) \Rightarrow p(\vec{x}) \qquad \mathcal{R} \Vdash I(\vec{x}) \Rightarrow \Box q(\vec{x}) \qquad \mathcal{R} \Vdash q(\vec{x}) \wedge p(\vec{x}) \Rightarrow \bigcirc p(\vec{x})}{\mathcal{R} \Vdash I(\vec{x}) \Rightarrow \Box p(\vec{x})} \text{ STR2}$$

$$\frac{\mathcal{E}_\Pi \Vdash \bigwedge_{\substack{(l \to r \text{ if } \gamma) \in R \\ (\theta, w, \gamma', \vec{v}) \in \Theta_{(l,r,\gamma)}}} p(r, \vec{v})\theta = \top \text{ if } \gamma\theta \wedge \gamma'\theta \wedge w\theta = \top \wedge q(l, \vec{v})\theta = \top}{\mathcal{R} \Vdash q(\vec{x}) \wedge p(\vec{x}) \Rightarrow \bigcirc p(\vec{x})} \text{ NR2}$$

Figure 4.3: Strengthenings for $\mathcal{R} = (\Sigma, E, R)$, with Θ as defined in Theorem 10.

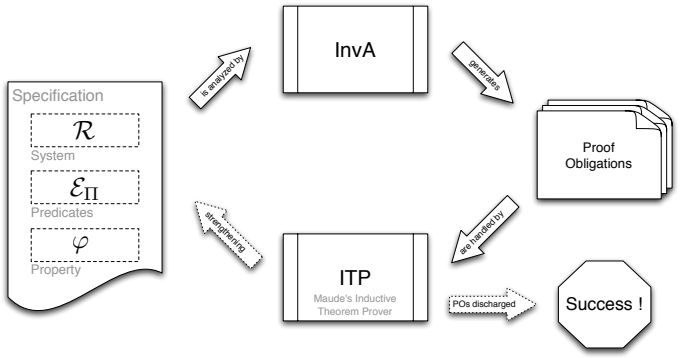

Figure 4.4: Approach for checking ground invariance and ground stability of rewrite theories.

are all conditional, have no variables in common with the rewrite rule, and have Ω-patterns as left-hand sides in the state parameter. If

$$\Theta_{(l,r,\gamma)} = \bigcup_{(p(v,v_1,\ldots,v_n) = w \text{ if } \gamma') \in E^p_\Pi} \{(\theta, w, \gamma', v_1, \ldots, v_n) \mid \theta \in CSU_B(v = l)\}.$$

then, inference rule NR2 *in Figure 4.3 is sound.*

Proof. Similar to the proof of Theorem 7. □

4.4 InvA: The Maude Invariant Analyzer Tool

The approach for proving ground stability properties in the InvA tool is depicted in Figure 4.4.

For a topmost rewrite theory $\mathcal{R}$ and of a set of state predicates Π specified in Maude, the InvA tool mechanizes inference rules St, Inv, Str1, Str2, Nr1, and Nr2. Given a ground stability or ground invariance property φ, it generates equational proof obligations such that, if they hold, then $\mathcal{T}_{\mathcal{R}} \models \varphi$. It also supports proving properties of the form $q \Rightarrow p$. Thanks to the availability since Maude 2.6 of unification modulo commutativity (C), associativity and commutativity (AC), and modulo these theories plus identities (U), and to the narrowing modulo infrastructure, the InvA tool can handle modules with operators declared C, CU, AC, and ACU.

4.4.1 Commands Available to the User

The commands available in the InvA tool are the following:

`(help .)` shows the list of commands available in the tool.

`(analyze-stable <pred> in <module> <module> .)` generates the proof obligations for proving the premise of inference St with inference Nr1, for the given predicate and the given modules. The first module equationally specifies the state predicate and the second one the topmost rewrite theory. This command tries to eagerly discharge the proof obligations; those that cannot be discharged are shown to the user.

`(analyze-stable <pred> in <module> <module> assuming <pred> .)` generates the proof obligations for proving the third premise of inference Str2 with inference Nr2, for the given predicate and the given modules. The first module equationally specifies the state predicates and the second one the topmost rewrite theory. This command tries to eagerly discharge the proof obligations; those that cannot be discharged are shown to the user.

`(analyze <pred> implies <pred> in <module> .)` generates the proof obligations for proving the given implication in the given module, according to inference C⇒. This command tries to eagerly discharge the proof obligations; those that cannot be discharged are shown to the user.

`(show pos .)` shows the proof obligations computed in the last `analyze` command that could not be discharged; those that were discharged are not shown.

`(show-all pos .)` shows all the proof obligations computed in the last `analyze` command.

Observe that the analysis commands in InvA give direct tool support for deductive reasoning with *some* of the inference rules presented in this chapter, but not for all of them. For example, there is no command in InvA directly supporting deduction with inference rule Inv. Nevertheless, deduction with *all* inference rules in this chapter is supported by InvA via *combination of commands*. For example, deduction with inference rule Inv can be achieved by combining the `analyze` and `analyze-stable` commands.

4.4.2 Automatic Discharge of Proof Obligations

After applying rules St, Inv, Str1, Str2, Nr1, and Nr2 according to the user commands, the InvA tool uses rewriting-based reasoning and narrowing procedures, and SMT decision procedures for automatically discharging as many of the generated equational proof obligations as possible. For an admissible equational specification $\mathcal{E} = (\Sigma, E \cup B)$ and a conditional proof obligation φ of the form

$$t = u \textbf{ if } \gamma,$$

the InvA tool applies a proof-search strategy such that, if it succeeds, then the Kripke structure associated to the initial reachability model satisfies φ. Otherwise, if the proof-search fails, the proof obligation φ (or an equivalent variant) is output to the user.

For the proof-search process, the InvA tool first tries to simplify Boolean expressions in φ and assumes that any operator '$\sim$' is an equationally defined equality predicate, i.e., an equality enrichment (see Section 2.3). Using this information, a Boolean transformation can be applied recursively to φ with the additional information of the equality enrichment, if any is defined. The goal of the Boolean transformation process is to obtain, if possible, an inductively equivalent proof obligation φ' for which the of automatic search techniques, explained below, have better chances of success.

The following is a description of the Boolean transformations applied recursively by the tool:

- If $t = u$ in φ is such that t is of the form $t_1 \sim t_2$ and u of the form $\bot$, then φ is transformed into $\top = \bot$ **if** $\gamma \wedge t_1 = t_2$.

- If $v_1 = v_2$, with $v_1, v_2 \in T_\Sigma(X)_{Bool}$, is any of the Σ-equalities in the condition γ of φ, then:
  - If v_1 is of the form $v_1^1 \sim v_1^2$ and v_2 of the form $\top$, then $v_1 = v_2$ is replaced by $v_1^1 = v_1^2$.
  - If v_1 is of the form $v_1^1 \sqcap \cdots \sqcap v_1^n$ and v_2 of the form $\top$, then $v_1 = v_2$ is replaced by $v_1^1 = \top \wedge \cdots \wedge v_1^n = \top$. Note that the v_1^i have sort *Bool*.
  - If v_1 is of the form $v_1^1 \sqcup \cdots \sqcup v_1^n$ and v_2 of the form $\bot$, then $v_1 = v_2$ is replaced by $v_1^1 = \bot \wedge \cdots \wedge v_1^n = \bot$. Note that the v_1^i have sort *Bool*.

Recall that $\sqcap$ and $\sqcup$ are the conjunction and disjunction function symbols used by the equality enrichment introduced in Section 2.3. Also note that Σ-equalities are unoriented, and thus in the Boolean transformation the order of terms in the equalities is immaterial.

After the Boolean transformation is completed, the following strategy is applied to the resulting proof obligation. Assume φ has been already simplified by the above transformation. Let $\overline{t}$, $\overline{u}$, $\overline{\gamma}$ be obtained by replacing each variable $x \in X$ by a new constant $\overline{x} \in \overline{X}$, with $\Sigma \cap \overline{X} = \varnothing$.

Equational simplification. First, the strategy checks if φ holds *trivially*, i.e., if $t \downarrow_{\Sigma,E/B} =_B u \downarrow_{\Sigma,E/B}$ or there is $t_i = u_i$ in γ such that $t_i \downarrow_{\Sigma,E/B}, u_i \downarrow_{\Sigma,E/B} \in T_\Sigma$ but $t_i \downarrow_{\Sigma,E/B} \neq_B u_i \downarrow_{\Sigma,E/B}$. Some simplifications in the form of reduction to canonical forms can be made to φ, even if they do not yield a trivial proof of φ. In some cases, such canonical reductions are incorporated into φ and the Boolean transformation used again.

Context joinability. Second, it checks whether φ is *context-joinable* [33]. The proof obligation φ is context-joinable if $\overline{t}$ and $\overline{u}$ are joinable in the rewrite theory $\mathcal{R}_{\mathcal{E}}^{\varphi} = (\Sigma(\overline{X}), B, \overrightarrow{E} \cup \overrightarrow{\gamma})$, obtained by making variables into constants and by orienting the equations E as rewrite rules $\overrightarrow{E}$ and *heuristically* orienting each equality $t_i = u_i$ in γ as a sequent $\overline{t_i} \rightarrow \overline{u_i}$ in $\overrightarrow{\gamma}$.

Unfeasability. Third, it checks if the proof obligation is *unfeasible* [33]. The proof obligation φ is unfeasible if there is a conjunct $\overline{t_i} \rightarrow \overline{u_i}$ in $\overrightarrow{\gamma}$ and $v, w \in T_\Sigma(X)$ such that $\mathcal{R}_{\mathcal{E}}^{\varphi} \vdash \overline{t_i} \rightarrow \overline{v} \wedge \overline{t_i} \rightarrow \overline{w}$, $CSU_B(v = w) = \varnothing$, and v and w are strongly irreducible with $\overrightarrow{E}$ modulo B.

SMT Solving. Last, it checks if the proof obligation can be proved by an SMT decision procedure. The condition γ of the proof obligation φ is analyzed and, if possible, a subformula consisting only of arithmetic subexpressions is extracted. This subformula has the following property: if it is a contradiction, then γ is unsatisfiable. Therefore, if the SMT decision procedure answers that the input subformula is unsatisfiable, then, as in the previous test, φ is unfeasible.

Because of the admissibility assumptions on $(\Sigma, E \cup B)$, the first test of the strategy either succeeds or fails in finitely many equational rewrite steps. For the second and third tests, the strategy is not guaranteed to succeed or fail in finitely many rewrite steps because the oriented sequents $\overrightarrow{\gamma}$ can falsify the termination assumption. So, for these last two checks, InvA uses a bound on the depth of the proof-search.

4.5 Related Work and Concluding Remarks

Chandy and Misra [20] and Manna and Pnueli [68] pursued the idea of using a deductive methodology to prove the invariance properties of concurrent systems specified in imperative languages. The notion of stability was inspired by the definition of the *stable* predicate in [76]. A comprehensive account of the vast literature on deductive approaches for verifying invariants of concurrent systems is beyond the scope of the present work; the aim here is more modest, namely, the focus is on related work using rewriting techniques for the deductive verification of invariants.

Rusu [94] proposes an approach for verifying invariant properties of a (possibly infinite-state) concurrent system specified by an unconditional topmost rewrite theory, following the ideas of Bruni and Meseguer [18]. That approach consists in casting an invariance problem of the form $\mathcal{R} \Vdash I \Rightarrow \Box p$ as an inductive problem of an equational theory $\mathcal{M}(\mathcal{R}, I)$ in membership equational logic, an equational sublogic of rewriting logic, as follows: $\mathcal{R} \Vdash I \Rightarrow \Box p$ if and only if $\mathcal{M}(\mathcal{R}, I) \Vdash p(t) = \top$ for every ground term t of sort *Reachable*, and t has sort *Reachable* in $\mathcal{M}(\mathcal{R}, I)$ if and only if t is $\mathcal{R}$-reachable from I. The approach in [94] is complemented by bounded symbolic execution, achieved by narrowing modulo, so that a property can be symbolically tested before trying to prove it invariant. The key difference between this approach and the one in this chapter is that the proof obligations generated for proving $\mathcal{M}(\mathcal{R}, I) \Vdash p(t) = \top$ do not take advantage of p's equational definition, in contrast to the narrowing-based reasoning incorporated in the

inference system here. The approach in this section can benefit from using narrowing for the symbolic testing of state predicates, although more research is required for handling conditional rewrite theories.

Proof scores in the OTS/CafeOBJ method are used to prove invariant properties of concurrent systems specified by *observational transition systems* (OTS) [80]. This approach has been applied for verifying safety properties of large specifications, including communication protocols. The approach is to divide a formula stating an invariant property into reasonably smaller ones by exploiting properties of the Boolean operators, each of which is proved by writing proof scores (or proof obligations) to be discharged individually by equational rewriting. The main difference between the approach presented here and the OTS one is that proof scores are constructed and manipulated manually by the user, which adds considerable time to the verification process. The interesting idea of exploiting the properties of Boolean operators needs to be further studied and considered within the inference framework here.

Combinations of deductive and algorithmic techniques have also been proposed for proving temporal logic properties φ of a (possibly infinite-state) concurrent system specified by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$. Equational abstraction [74] reduces the problem of whether $\mathcal{R}$ satisfies φ to model checking φ on a finite state abstract version $\mathcal{R}/\Delta = (\Sigma, E \cup E_\Delta, R \cup R_\Delta)$. Invisible transitions [42] approach the problem of whether $\mathcal{R}$ satisfies φ by identifying a subset $S \subseteq R$ of rewrite rules that are φ-invisible (i.e., rewriting with S does not change the truth value of the predicate φ) to model checking that property on a finite state simplified version $\mathcal{R}/S = (\Sigma, E \cup S, R \setminus S)$. Both equational abstractions and invisible transition techniques tackle the verification problem of infinite-state systems by making finite the state space explosion, so that model checking methods are decidable. These two approaches, as it is also the case in the approach discussed here, require user-intervention for defining, respectively, the abstraction predicates and the invisible rewrite rules, and for discharging the inductive proof obligations resulting from the corresponding transformations (i.e., admissibility conditions plus the proof obligations specific to each method). In particular, the checking algorithms based on narrowing presented in this chapter can be used to generate proof obligations for checking the rewrite rules $S \subseteq R$ of $\mathcal{R}$, p-invisible for a state predicate p. These approaches can complement each other and can be combined, resulting in a powerful and versatile framework for proving temporal properties of rewrite theories. The mechanization of these three approaches in order to reduce user intervention is an exciting

topic for further research.

Narrowing-based symbolic model checking techniques for topmost rewrite theories $\mathcal{R}$ have been previously studied in [38], where the idea is to "fold" the narrowing tree for $\mathcal{R}$ that can in practice result in finite-state system that symbolically simulates $\mathcal{R}$. It is worth pursuing an extension of these narrowing symbolic model checking techniques for conditional rewrite theories, so that the two approaches can be combined for symbolic model checking and for symbolic simulation (following the idea of Rusu in [94]).

CHAPTER 5

INVA CASE STUDY I: RELIABLE COMMUNICATION IN THE ALTERNATING BIT PROTOCOL

This chapter presents the first case study about the deductive analysis of inductive safety properties using the methodology, the proof system, and the Maude Invariant Analyzer tool (InvA) introduced in Chapter 4. The subject of study is a highly concurrent protocol for reliable data communication across a lossy channel. As a result, this chapter reports on the successful and full mechanical verification in the InvA tool of a main invariant for the communication protocol.

Implementations of the message-passing model, such as TCP/IP, provide the programmer with an abstraction of a stream of data messages that communicate two parties such that each message is delivered in the order sent. However, the physical communication between these two parties is not necessarily reliable. Due to congestion in the network (other traffic), transient noise, buffer overflows, or other problems, some of the messages sent out over the network may not actually arrive to the other end, or they may arrive incomplete or corrupted (usually detected with some sort of checksum). Furthermore, messages may arrive out of order. Therefore, it is necessary to include in these message-passing implementations a protocol that ensures that when a message is lost, it is retransmitted by the sender. At the same time, the protocol must guarantee that duplicate messages are not delivered to the receiver and that the delivery order is consistent with the sending order.

The *Alternating Bit Protocol* (ABP) [8], is a simple, yet effective, protocol for managing the retransmission of lost or corrupted messages. The protocol works by identifying two processes, the sender and receiver, each with a control bit called the *alternating bit*, using a lossy channel. Both

sender and receiver can send messages across the channel and every message sent is signed with the current bit. By using the alternating bits, the protocol can successfully identify message loss or corruption. In the proof technologies that address distributed concurrent non-deterministic systems, ABP is a well-established benchmark and it is perhaps the simplest non-trivial example of such a system. This chapter uses ABP as the basis of the analysis.

The invariant in this case study is about reliable communication in ABP, which is the main safety property of the protocol. As a result of the case study, a fully mechanized proof for the correctness of the protocol is obtained with the InvA tool, and with help of Maude's ITP that was useful for discharging some equational proof obligations and auxiliary lemmata. The proof relies heavily on the specification and verification methods of Chapter 4, and their implementation in the InvA tool.

This chapter is organized as follows. A summary of ABP and the modeling methodology in Maude are explained in Section 5.1. The discussion on the verification task for the reliability property of ABP is documented in Section 5.2. Related work and a comparison with other formal verification case studies regarding the protocol are collected in Section 5.3. The formal specification of ABP and a proof of its admissibility, including the specification of state predicates, auxiliary functions, auxiliary lemmata, and ITP proof scripts can be found in Appendix B.

5.1 ABP

The *Alternating Bit Protocol* (ABP) [8] is a data layer protocol. It was designed to achieve reliable full-duplex data transfer between two processes over an unreliable half-duplex transmission line in which messages can be lost or corrupted in a detectable way. The data link layer, the second lowest layer in the OSI seven layer model, splits data into frames for sending on the physical layer and receives acknowledgment frames. It performs error checking and re-transmits frames not received correctly. It provides an error-free virtual channel to the network layer, the third lowest layer in the OSI layer model.

The overall structure of ABP is illustrated in Figure 5.1. The protocol comprises an input stream of data to be transmitted, a *sender* and a *receiver* process, each having a data buffer and a one *bit* state, a *data channel* for data-bit pairs called *bit-packets*, an *acknowledgment channel* for bit-packets

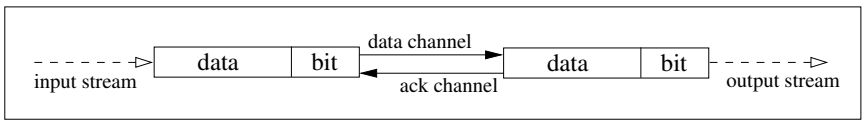

Figure 5.1: The Alternating Bit Protocol.

consisting of a single bit, and an output data stream. Here is how the protocol works:

- The sender process starts by repeatedly sending bit-packets (b, d_1) into the data channel, where b is the sender's bit and d_1 is the first element of the input stream.
- The receiver process starts by waiting until it receives the bit-packet (b, d_1), and then it repeatedly sends b over the acknowledgment channel.
- When the source process receives b, it begins repeatedly sending the bit-packet $(\mathit{flip}(b), d_2)$, where d_2 is the second element of the input stream, which is what the receiver process is now waiting for.
- When the target receives $(\mathit{flip}(b), d_2)$, it begins sending packets containing $\mathit{flip}(b)$.
- At any moment either channel can duplicate or lose its oldest packet, if any.
- And so on ...

The protocol is highly concurrent and non-deterministic because, for instance, it is unknown how long will it take before a bit-packet gets through. To guarantee progress, it must be assumed that the channels are fair, in the sense that if the sender persists, eventually a bit-packet will get through. The reason is that without this assumption the algorithm is not correct because data transmission might fail forever. However, this is a fairness assumption that is not needed for analyzing the reliable communication enforced by the protocol. Remember that a safety property assures that "nothing bad happens", even when nothing ever happens.

5.1.1 Formal Modeling

The ABP specification in Maude has 9 modules. This section should be read in connection with Appendix B, which contains the full formal specification.

At the top level, the state space is represented by the top sort `Sys` defined in module `ABP-STATE`, which is a 6-tuple:

```
sort Sys .
op _:_>_|_<_:_ : iNat Bit BitPacketQueue BitQueue Bit iNatList
                 -> Sys [ctor] .
```

The arguments of a state are the data from the input stream currently being transmitted by the sender (as `iNat`), the bit of the sender (as `Bit`), the data channel (as `BitPacketQueue`), the acknowledgment channel (as `BitQueue`), the bit of the receiver (as `Bit`), and the output stream (as `iNatList`).

The sort `iNat` is that of natural numbers in Peano notation, together with an equality enrichment. Natural numbers are used to represent packets in the potentially infinite input stream.

```
sort iNat .
op 0 : -> iNat [ctor] .
op s_ : iNat -> iNat [ctor] .
op _~_ : iNat iNat -> Bool [comm] .
```

Bits are defined in module `BIT` by sort `Bit` with two constructor constants, a 'flipping' operator, and an equality enrichment:

```
sort Bit .
ops on off : -> Bit [ctor] .
op flip : Bit -> Bit .
op _~_ : Bit Bit -> Bool [comm] .

eq flip(on)
 = off .
eq flip(off)
 = on .
```

Sort `BitPacketQueue` represents lists of bit-packets, sort `BitQueue` represents lists of bits, and sort `iNatList` represents lists of natural numbers. They are all lists defined in the usual way: an empty list is identified by the constructor constant `nil`, "cons" is a constructor binary symbol denoted by juxtaposition, and append is a defined binary symbol denoted by '`;`'. For instance, sort `BitQueue` defined in module `BIT-QUEUE` is specified as follows:

```
sort BitQueue .
op nil : -> BitQueue [ctor] .
op __ : Bit BitQueue -> BitQueue [ctor prec 61] .
op _;_ : BitQueue BitQueue -> BitQueue [prec 65] .

eq nil ; BQ:BitQueue
 = BQ:BitQueue .
eq B1:Bit BQ1:BitQueue ; BQ2:BitQueue
```

```
 = B:Bit (BQ1:BitQueue ; BQ2:BitQueue) .
```

Having covered the basic notation, consider the following ground term of sort `Sys` representing a state in the system:

```
s(0)  :  on  >  (off,0) nil  |  nil  <  off  :  (0 nil)
```

In this state, the packet from the input stream currently being sent in `s(0)`, the sender's bit is `on`, the data channel contains only the bit-packet `(off,0)`, the acknowledgment channel is empty, the receiver's bit is `off`, and the output stream consists only of the packet `0`.

Finally, module `ABP` specifies the operation of the protocol with 15 rewrite rules. These rewrite rules model the transmission of the bit-packets through the data channel, the reception of acknowledgments from the receiver, data duplication and loss, among other behaviors of the system. For instance, consider the following five rewrite rules:

```
rl [send-1] :
   N:iNat : B1:Bit > BPQ:BitPacketQueue
                   | BQ:BitQueue < B2:Bit : NL:iNatList
=> N:iNat : B1:Bit > BPQ:BitPacketQueue ; ((B1:Bit, N:iNat) nil)
                   | BQ:BitQueue < B2:Bit : NL:iNatList .

rl [recv-1b] :
   N:iNat : on     > BPQ:BitPacketQueue
                   | off BQ:BitQueue < B2:Bit : NL:iNatList
=> s(N:iNat) : off > BPQ:BitPacketQueue
                   | BQ:BitQueue < B2:Bit : NL:iNatList .

rl [recv-1c] :
   N:iNat : off    > BPQ:BitPacketQueue
                   | on BQ:BitQueue < B2:Bit : NL:iNatList
=> s(N:iNat) : on >  BPQ:BitPacketQueue
                   | BQ:BitQueue < B2:Bit : NL:iNatList .

rl [recv-2a] :
   N:iNat : B1:Bit > (on,N2:iNat) BPQ:BitPacketQueue
                   | BQ:BitQueue < on : NL:iNatList
=> N:iNat : B1:Bit > BPQ:BitPacketQueue
                   | BQ:BitQueue < off : (N2:iNat NL:iNatList) .

rl [dup-1] :
   N:iNat : B1:Bit > BP:BitPacket BPQ:BitPacketQueue
                   | BQ:BitQueue < B2:Bit : NL:iNatList
=> N:iNat : B1:Bit > BP:BitPacket (BP:BitPacket BPQ:BitPacketQueue)
                   | BQ:BitQueue < B2:Bit : NL:iNatList .
```

The effects of these rules in a state can be summarized as follows:

`[send-1]` models the "fifo" placement of the current bit-packet in the data channel (the acknowledgment channel behaves in the same way).

`[recv-1b]` models the reception of the acknowledgment the sender was waiting for and thus the sender process immediately updates the packet to be transmitted with the next available packet from the input stream and flips its communication bit.

`[recv-1c]` models the reception of an acknowledgment the sender was not waiting for and thus the acknowledgment is ignored.

`[recv-2a]` models the reception of a bit-packet whose contents are put in the output stream.

`[dup-1]` duplicates the first message in the data channel.

Note that because of rule `[recv-1c]`, for instance, the formal model of the ABP has potentially infinitely many reachable states: every time a packet is successfully transmitted, the sender's counter modeling the input stream is increased by one and then the whole sending process starts over again but the next packet.

5.2 Reliable Communication

The analysis that follows is based on the formal model explained in Section 5.1.1.

One of the main properties the ABP should enjoy is the reliable communication property. This means that the protocol makes possible to *reliably* communicate and deliver information from a source to a destination, even in the presence of unreliable channels of communication. The goal in this section is to report on the experience of using the InvA tool in the successful and mechanical verification of this property.

5.2.1 Formal Specification of the Property

Reliable communication in `ABP` means that whenever n packets have been delivered, these were the first n packets sent in that particular order. Note that this is a property that must hold for each natural number n and that cannot be effectively checked by means of direct algorithmic techniques, such as model checking the `ABP` specification, even if the set of initial states is finite.

The reliable communication property is expressed by the state predicate `inv-main` and is defined as follows:

```
op inv-main : Sys -> Bool .

eq [inv-main-1] :
   inv-main(N:iNat : B:Bit  > BPQ:BitPacketQueue
                            | BQ:BitQueue < B:Bit : NL:iNatList)
 = (N:iNat NL:iNatList) ~ gen-list(N:iNat) .
ceq [inv-main-2] :
    inv-main(N:iNat : B1:Bit > BPQ:BitPacketQueue
                            | BQ:BitQueue < B2:Bit : NL:iNatList)
 = NL:iNatList ~ gen-list(N:iNat)
 if B1:Bit ~ B2:Bit = false .

op gen-list : iNat -> iNatList .

eq gen-list(0)
 = (0 nil) .
eq gen-list(s N)
 = (s N) gen-list(N) .
```

State predicate `inv-main` is fully defined by two equations and uses the auxiliary function `gen-list`. Equation `[inv-main-1]` considers the case in which the parity of the sender and receiver bits coincides. In this case, the reliable communication property holds if and only if the delivered packets correspond to all but the last packet sent and they are all in order. Equation `[inv-main-2]` considers the case in which the parity of the sender and receiver bits does not coincide. In this case, the reliable communication property holds if and only if the delivered packets correspond to all packets sent and they are all in order. Given a natural number n, function `gen-list` generates the list of the first n natural numbers in decreasing order.

Consider the rule `[recv-2b]` that models packet reception in `ABP` in order to motivate the correctness of the reliable communication property:

```
rl [recv-2b] :
   N:iNat : B:Bit > (off,N1:iNat) BPQ:BitPacketQueue
                  | BQ:BitQueue < off : NL:iNatList
=> N:iNat : B:Bit > BPQ:BitPacketQueue
                  | BQ:BitQueue < on : (N1:iNat NL:iNatList) .
```

Note that when a packet `N1:iNat` is received, there is no assumption made on the relationship between `N1:iNat` and the current packet from the input stream `N:iNat` or the already delivered packets `NL:iNatList`. In this case, there is no obvious reason for the reliable communication property to hold, even if a state initially satisfies this property.

The goal is to prove the `ABP` `inv-main`-invariant from `init`. State predicate `init` defines the set of initial states as follows:

```
op init : Sys -> [Bool] .
```

```
  eq [init-1] :
     init( 0 : on > nil | nil < on : nil)
   = true .
  eq [init-2] :
     init( 0 : off > nil | nil < off : nil)
   = true .
```

The set of initial states for the verification task at hand, as defined by `init`, consists of exactly two states. Namely, those states where the packet to be transmitted is `0`, the sender and receiver bits coincide, the communication channels are empty, and no packet has been delivered.

The following verification commands can be given to the InvA tool in order to check if state predicate `inv-main` is an inductive invariant from `init`:

```
(analyze init(S:Sys) implies inv-main(S:Sys) in ABP-PREDS .)

(analyze-stable inv-main(S:Sys) in ABP-PREDS ABP .)
```

It is assumed that module `ABP-PREDS` contains the state predicates and their corresponding auxiliary function symbols, and module `ABP` contains the specification of ABP, as explained in Section 5.1.1 and documented in Appendix B.

When issuing the above-mentioned commands, the InvA tool generates the following output:

```
Checking ABP-PREDS ||- init(S:Sys) => inv-main(S:Sys) ...
Proof obligations generated:  2
Proof obligations discharged: 2
Success!

Checking ABP-PREDS ||- inv-main(S:Sys) => 0 inv-main(S:Sys) ...
Proof obligations generated:  30
Proof obligations discharged: 22
The following proof obligations need to be discharged:
8. from inv-main-2 & recv-2b : pending
    inv-main(#7:iNat : #8:Bit > #10:BitPacketQueue
      | #11:BitQueue < on :(#9:iNat #12:iNatList)) = true
 if off ~ #8:Bit = false
 /\ #12:iNatList = gen-list(#7:iNat).
...
```

The tool generates 32 proof obligations and automatically discharges 24 of them. The remaining 8 proof obligations are returned to the user; in the snapshot, only one proof obligation for ground stability that was not automatically discharged is shown and it is identified by label 8.

Upon inspection of the InvA's output, it is relatively easy to observe that `inv-main` is not an inductive invariant for `ABP`. Indeed, consider the proof obligation identified by label 8, as show in the snapshot above, and

a ground interpretation where `#8:Bit` is `on`, `#7:iNat` and `#9:iNat` are `0`, and `#12:iNatList` is the singleton list `0 nil`. For this particular ground instantiation, the condition in the proof obligation is satisfied because `on ~ off` reduces to `false` and the value returned by `gen-list` on input `0` is the ground list `0 nil`. However, by equation `[inv-main-2]` in the definition of predicate `inv-main`, this proof obligation is false because the lefthand side of the conclusion reduces to the Boolean term `0 nil ~ 0 0 nil`, which ultimately reduces to `false`. This is evidence of the fact that a stronger predicate is needed, that is, `inv-main` needs to be strengthened.

5.2.2 Strengthening the Invariant

The first observation to make is that the InvA tool would be able to automatically discharge more proof obligations and also return simpler ones, if there was some mechanism for achieving case analysis on the sort `Bit`. Since the InvA internals do not offer this feature yet, a practical approach is to include the case splitting as part of the predicate's equational definition (similarly to what was done in the definition of state predicate `init`). For instance, state predicate `inv` is a finer-grained version of `inv-main` that exhibits the idea of case splitting on the sort `Bit` for the case of the bits in the sender and receiver.

```
op inv : Sys -> Bool .

eq [inv-1a] :
   inv(N:iNat : on  > BPQ:BitPacketQueue
                    | BQ:BitQueue < on : NL:iNatList)
 = (N:iNat NL:iNatList) ~ gen-list(N:iNat) .
eq [inv-1a] :
   inv(N:iNat : off > BPQ:BitPacketQueue
                    | BQ:BitQueue < off : NL:iNatList)
 = (N:iNat NL:iNatList) ~ gen-list(N:iNat) .
eq [inv-2a] :
   inv(N:iNat : on  > BPQ:BitPacketQueue
                    | BQ:BitQueue < off : NL:iNatList)
 = NL:iNatList ~ gen-list(N:iNat) .
eq [inv-2a] :
   inv(N:iNat : off > BPQ:BitPacketQueue
                    | BQ:BitQueue < on : NL:iNatList)
 = NL:iNatList ~ gen-list(N:iNat) .
```

Since the case analysis on the sort `Bit` is already implemented in predicate `inv`, and this is potentially useful for automation in the overall proof, this predicate is preferred over predicate `inv-main`. The idea is then to strengthen `inv` instead of `inv-main`. Within the overall context of the ver-

ification task, the change of predicate `inv-main` for `inv` requires a formal proof of the following implications:

$$\texttt{ABP} \Vdash \texttt{init} \Rightarrow \texttt{inv} \qquad \text{and} \qquad \texttt{ABP} \Vdash \texttt{inv} \Rightarrow \texttt{inv-main}.$$

These two proof obligations can be analyzed with the help of inference rule C⇒ in Section 4.4. The InvA's mechanization of this inference rule can automatically discharge the implications:

```
Checking ABP-PREDS ||- init(S:Sys) => inv(S:Sys) ...
Proof obligations generated:  2
Proof obligations discharged: 2
Success!

Checking ABP-PREDS ||- inv(S:Sys) => inv-main(S:Sys) ...
Proof obligations generated:  4
Proof obligations discharged: 4
Success!
```

Finding a strengthening for `inv` is not an easy task at first sight. The non-obvious relationships between the channels and the alternating bits, and the many rules that can concurrently apply to a state make this harder. But it is the deep understanding of these relationships that guides the proof effort for obtaining a useful, yet succinct and elegant, strengthening for `inv`.

The key to it all is that the channels behave under some sort of uniformity that is parametric on the sender and receiver bits. This notion of uniformity can be precisely captured with the help of some auxiliary predicates for the two communication channels. Indeed, consider the following auxiliary predicates `all-packets` and `good-packet-queue`:

```
  op all-packets : BitPacketQueue Bit iNat -> Bool .

  eq [ap-1] :
     all-packets(nil,B:Bit,N:iNat)
   = true .
  eq [ap-2] :
     all-packets(BP:BitPacket BPQ:BitPacketQueue,B:Bit,N:iNat)
   = BP:BitPacket ~ (B:Bit,N:iNat) and
     all-packets(BPQ:BitPacketQueue,B:Bit,N:iNat) .

  op good-packet-queue : BitPacketQueue Bit iNat -> Bool .

  eq [gpq-1] :
     good-packet-queue(nil,B:Bit,N:iNat)
   = true .
 ceq [gpq-2] :
     good-packet-queue((B1:Bit,N1:iNat) BPQ:BitPacketQueue,
                       B:Bit,N:iNat)
```

```
   = N:iNat ~ s(N1:iNat) and
     good-packet-queue(BPQ:BitPacketQueue,B:Bit,N:iNat)
  if B1:Bit = flip(B:Bit) .
  eq [gpq-3] :
     good-packet-queue((B:Bit,N1:iNat) BPQ:BitPacketQueue,
                       B:Bit,N:iNat)
   = N:iNat ~ N1:iNat and
     all-packets(BPQ:BitPacketQueue,B:Bit,N:Nat) .
```

Predicate `all-packets` on input `BPQ:BitPacketQueue` and `(B:Bit,N:iNat)` is true if and only if all bit-packets in `BPQ` have the form `(B,N)`. Predicate `good-packet-queue` on input `BPQ:BitPacketQueue` and `(B:Bit,N:iNat)` is true if and only if `BPQ` can be split into two parts, one of them possibly empty, where in the initial part of the channel all packets are of the form `(flip(B),N-1)` and in the second part of the form `(B,N)`. For example:

```
good-packet-queue((on,3) (off,4) (off,4) nil, off, 4) = true
good-packet-queue((on,3) (on,3) nil, off, 4) = true
good-packet-queue((off,4) nil, off, 4) = true
good-packet-queue((off,4) (on,4) nil, off, 4) = false
```

Auxiliary predicates `all-bits` and `good-bit-queue` are similar to the auxiliary predicates just discussed for channels of bit-packets, but they are about channels of bits.

```
  op all-bits : BitQueue Bit -> Bool .

  eq [ab-1] :
     all-bits(nil,B:Bit)
   = true .
  eq [ab-2] :
     all-bits(B1:Bit BQ:BitQueue,B:Bit)
   = B1:Bit ~ B:Bit and all-bits(BQ:BitQueue,B:Bit) .

  op good-bit-queue : BitQueue Bit -> Bool .

  eq [gbq-1] :
     good-bit-queue(nil,B:Bit)
   = true .
 ceq [gbq-2] :
     good-bit-queue(B1:Bit BQ:BitQueue, B:Bit)
   = good-bit-queue(BQ:BitQueue,B:Bit)
  if B1:Bit = flip(B:Bit) .
  eq [gbq-3] :
     good-bit-queue(B:Bit BQ:BitQueue, B:Bit)
   = all-bits(BQ:BitQueue,B:Bit) .
```

The strengthening for `inv` is the state predicate `good-queues` that uses the auxiliary predicates above-mentioned:

```
  op good-queues : Sys -> Bool .
```

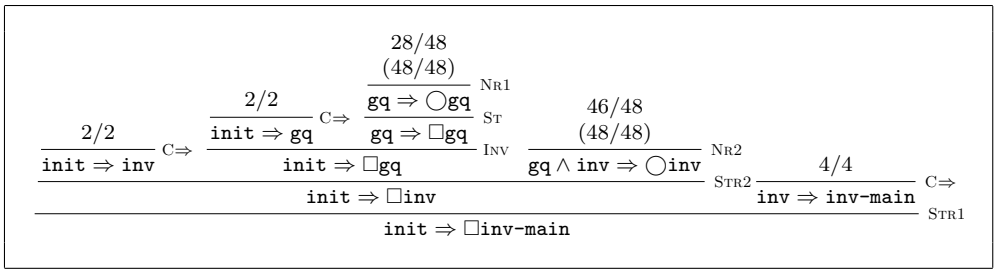

Figure 5.2: Correctness proof of the Alternating Bit Protocol (`gq` stands for `good-queues`). The expression d/g denotes the number g of proof obligations generated and the number d of proof obligations automatically discharged by the InvA tool; the same expression in parenthesis has the same meaning but includes the use of the ITP and/or some auxiliary lemmata. Some trivial inferences have been omitted.

```
eq [good-queues-1a] :
   good-queues(N:iNat : on > BPQ:BitPacketQueue |
               BQ:BitQueue < on : NL:iNatList)
 = all-bits(BQ:BitQueue,on) and
   good-packet-queue(BPQ:BitPacketQueue,on,N:iNat) .
eq [good-queues-1b] :
   good-queues(N:iNat : off > BPQ:BitPacketQueue |
               BQ:BitQueue < off : NL:iNatList)
 = all-bits(BQ:BitQueue,off) and
   good-packet-queue(BPQ:BitPacketQueue,off,N:iNat) .
eq [good-queues-2a] :
   good-queues(N:iNat : on > BPQ:BitPacketQueue |
               BQ:BitQueue < off : NL:iNatList)
 = good-bit-queue(BQ:BitQueue,off) and
   all-packets(BPQ:BitPacketQueue,on,N:iNat) .
eq [good-queues-2b] :
   good-queues(N:iNat : off > BPQ:BitPacketQueue |
               BQ:BitQueue < on : NL:iNatList)
 = good-bit-queue(BQ:BitQueue,on) and
   all-packets(BPQ:BitPacketQueue,off,N:iNat) .
```

State predicate `good-queues` is fully defined by 4 equations. It characterizes the patterns observed on the communication channels, and their relationship with the alternating bits.

As it will be shown, the strengthening `good-queues` of `inv` is enough to prove the correctness of ABP. Figure 5.2 depicts the full proof-tree for the ground invariance of `inv-main` from `init` that uses state predicates `inv` and `good-queues`.

The next step in the prove is to check

$$\texttt{ABP} \Vdash \texttt{good-queues} \wedge \texttt{inv} \Rightarrow \bigcirc \texttt{inv} \quad \text{and}$$

$$\texttt{ABP} \Vdash \texttt{init} \Rightarrow \Box \texttt{good-queues},$$

since the following two properties have been already proved:

$$\texttt{ABP} \Vdash \texttt{init} \Rightarrow \texttt{inv} \quad \text{and} \quad \texttt{ABP} \Vdash \texttt{inv} \Rightarrow \texttt{inv-main}.$$

When checking $\texttt{good-queues} \wedge \texttt{inv} \Rightarrow \bigcirc\texttt{inv}$, the following is the output given by the InvA tool:

```
rewrites: 97315 in 348ms cpu (346ms real) (279623 rewrites/second)
Checking ABP-PREDS ||- inv(S:Sys) => O inv(S:Sys)
  assuming good-queues(S:Sys) ...
Proof obligations generated:  48
Proof obligations discharged: 46
The following proof obligations could not be discharged:
8. from inv-1a & recv-2b : pending
    gen-list(#5:iNat)~(#6:iNat #9:iNatList) = true
 if #5:iNat = #6:iNat
 /\ all-bits(#8:BitQueue,off) = true
 /\ all-packets(#7:BitPacketQueue,off,#5:iNat) = true
 /\ gen-list(#5:iNat) = #5:iNat #9:iNatList .
46. from inv-1a & recv-2a : pending
    gen-list(#5:iNat)~(#6:iNat #9:iNatList) = true
 if #5:iNat = #6:iNat
 /\ all-bits(#8:BitQueue,on) = true
 /\ all-packets(#7:BitPacketQueue,on,#5:iNat) = true
 /\ gen-list(#5:iNat) = #5:iNat #9:iNatList .
```

The tool generates 48 proof obligations and automatically discharges 46 of them. The remaining 2 proof obligations are about properties of lists of natural numbers. Note that the Boolean transformation internally implemented by the InvA tool (explained in Section 4.4) splits the Boolean conjunctions in the specification of good-queues into conditions and the equality predicate '$\sim$' into '=', whenever it was possible. A proof script for proof obligations 8 and 46, that automatically discharges these proof obligations, can be given to the ITP as follows:

```
(goal po8 : ABP-PREDS |- A{ #5:iNat ; #6:iNat ; #9:iNatList ;
                            #8:BitQueue ; #7:BitPacketQueue }
 (
  (#5:iNat) = (#6:iNat) &
  (all-bits(#8:BitQueue,off)) = (true) &
  (all-packets(#7:BitPacketQueue,off,#5:iNat)) = (true) &
  (gen-list(#5:iNat)) = (#5:iNat #9:iNatList)
  =>
  (gen-list(#5:iNat) ~ (#6:iNat #9:iNatList)) = (true)
  )
.)
(auto .)

(goal po46 : ABP-PREDS |- A{ #5:iNat ; #6:iNat ; #9:iNatList ;
                             #8:BitQueue ; #7:BitPacketQueue }
```

```
 (
  (#5:iNat) = (#6:iNat) &
  (all-bits(#8:BitQueue,on)) = (true) &
  (all-packets(#7:BitPacketQueue,on,#5:iNat)) = (true) &
  (gen-list(#5:iNat)) = (#5:iNat #9:iNatList)
  =>
  (gen-list(#5:iNat) ~ (#6:iNat #9:iNatList)) = (true)
  )
.)
(auto .)
```

The following is the output of the ITP:

```
==================================
label-sel: po8#0@0
==================================
A{#5:iNat ; #6:iNat ; #7:BitPacketQueue ; #8:BitQueue ; #9:iNatList}
 gen-list(#5:iNat) = #5:iNat #9:iNatList
 & all-packets(#7:BitPacketQueue,off,#5:iNat) = true
 & all-bits(#8:BitQueue,off) = true & #5:iNat = #6:iNat
 ==> gen-list(#5:iNat)~(#6:iNat #9:iNatList) = true

++++++++++++++++++++++++++++++++++

rewrites: 10751 in 173ms cpu (181ms real) (61990 rewrites/second)
Eliminated current goal.

q.e.d

++++++++++++++++++++++++++++++++++

rewrites: 9172 in 51ms cpu (51ms real) (177962 rewrites/second)

==================================
label-sel: po46#1@0
==================================
A{#5:iNat ; #6:iNat ; #7:BitPacketQueue ; #8:BitQueue ; #9:iNatList}
 gen-list(#5:iNat) = #5:iNat #9:iNatList
 & all-packets(#7:BitPacketQueue,on,#5:iNat) = true
 & all-bits(#8:BitQueue,on) = true & #5:iNat = #6:iNat
 ==> gen-list(#5:iNat)~(#6:iNat #9:iNatList) = true

++++++++++++++++++++++++++++++++++

rewrites: 10751 in 179ms cpu (182ms real) (59745 rewrites/second)
Eliminated current goal.

q.e.d

++++++++++++++++++++++++++++++++++
```

This completes the proof of:

$$\texttt{ABP} \Vdash \texttt{good-queues} \wedge \texttt{inv} \Rightarrow \bigcirc\texttt{inv}.$$

For the proof of `init` $\Rightarrow \Box$`good-queues` the InvA tool gives the following output:

```
rewrites: 10072 in 32ms cpu (35ms real) (314730 rewrites/second)
Checking ABP-PREDS ||- init(S:Sys) => good-queues(S:Sys) ...
Proof obligations generated:  2
Proof obligations discharged: 2
Success!

rewrites: 57223 in 284ms cpu (283ms real) (201476 rewrites/second)
Checking
  ABP-PREDS+LEMMATA ||- good-queues(S:Sys) => O good-queues(S:Sys) ...
Proof obligations generated:  48
Proof obligations discharged: 48
Success!
```

Note that in the proof of ground stability, module `ABP-PREDS+LEMMATA` is used instead of `ABP-PREDS`. The former module contains 10 lemmata about the auxiliary predicates used by state predicate `good-queues`. Without these lemmata, the InvA tool discharges automatically only 26 of the 48 proof obligations. See Appendix B for a complete explanation of these lemmata and their mechanical proof in the ITP. This concludes the proof of the ground invariance of `good-queues` from `init` for `ABP`.

The main result about the correctness of the ABP is then established mechanically in the InvA with help of the ITP. Namely, the following inductive property holds:

$$\texttt{ABP} \Vdash \texttt{init} \Rightarrow \Box\texttt{inv-main}.$$

See Appendix B for mechanical proofs of the admissibility of modules `ABP`, `ABP-PREDS`, `ABP-PREDS+LEMMATA`, and also for the ITP proof scripts used as part of the main result in this chapter.

5.3 Related Work and Concluding Remarks

The Alternating Bit Protocol (ABP) is a well-established benchmark in the proof technologies that address concurrent, non-deterministic systems. As such, it has been formally studied from different viewpoints using a wealth of formal techniques. They include process algebra [10, 12], temporal Petri nets [100], the Calculus of Constructions [43], and timed rewriting logic [98], among many others.

In the framework of observational transition systems (OTS), ABP has been formally studied independently by K. Ogata and K. Futatsugi [81],

	Measure	[81]	This chapter
Model	LOC	286	208
Model + Predicates	LOC	286 + 63	208 + 200
State predicates	#	11	3
Lemmata	#	7	10
Proof scripts	LOC	5189	213
Proof scripts / # predicates	LOC	471.8	71

Figure 5.3: Comparison of the ABP case study for the reliable communication property with a similar case study using proof scores in [81].

and by K. Lin and J. Goguen [65]. In the former, the focus is on proving the same invariant property about reliable communication based on simultaneous induction. In the latter, the focus is on verifying liveness properties using conditional circular coinductive rewriting.

Figure 5.3 presents a comparison between the proof of the reliable communication property for ABP presented in [81], that uses proof scores, and the one presented here. This comparison is possible thanks to the authors of [81] who kindly shared the source code of their case study.

Note that the human proof effort in [81] is significantly higher than the one in proving the same property using the approach and tools of Chapter 4, as presented in this chapter. However, this comparison needs to be taken with a grain of salt, because many proof obligations in [81] are basic base cases and there is no tool support for discharging them. In contrast, the ITP was of great help, not only because it automatically took care of many simple proof obligations, but also because of some of its equational inductive techniques such as cover-set induction.

CHAPTER 6

INVA CASE STUDY II: SOME SAFETY PROPERTIES OF IBOS

From Chapter 4 it is known that a safety property of a software system asserts that "something bad should never happen". This chapter presents a case study on the mechanical verification of invariants for a browsing system designed to satisfy explicit security requirements. The main invariant studied here is of prime interest in its own right, but it is also intended as challenging benchmark for the inference system and its implementation in the InvA tool from Chapter 4. As a result, this chapter reports on the successful mechanical verification of some auxiliary invariants for the browsing system, that help in the verification task for the main invariant. It also reports on some limitations, in both the current methods and tool, that have restricted the degree of success in obtaining a full mechanical verification for the main invariant.

The notion of a browser that is to be secure by design is exemplified in the work on the *Illinois Browser Operating System* (IBOS) [101]. In contrast to current web browsers that have enormous trusted computing bases that commonly provide attackers with easy access to computer systems, IBOS drastically reduces the trusted computing base for web browsers. This chapter uses IBOS as the basis of the analysis and builds upon an updated version of a Maude specification previously developed by R. Sasse [95, 96] that is amenable for analysis in the InvA tool.

The main invariant is about the address bar being correct at all times, which is an important safety property for a browser. In contrast to the successful treatment of this similar property in [95, 96], in which reduction techniques were used to obtain a finitary and tractable search space, the specification in this chapter is used as it is for the purpose of the verifica-

tion task. As a result of the case study, low-level specification invariants, each literally generating thousands of proof obligations, are automatically handled by the tool. However, the same methods and techniques are partially limited when it comes to proving the address bar being correct. All in all, the IBOS specification is to date, and by far, the most challenging case study analyzed with the InvA tool.

A summary of IBOS and the modeling methodology in Maude are explained, respectively, in Section 6.1 and Section 6.2. The discussion on the verification task for the correctness of the address bar is documented in Section 6.3, as well as the mechanical verification of two auxiliary invariants. A summary of the limitations identified during the development of the case study are collected in Section 6.3.4.

6.1 IBOS

This section closely follows and adapts the introductory treatment of IBOS by R. Sasse [95], Section 4.1.

The Illinois Browser Operating System (IBOS) [101] is a web browsing system developed at the University of Illinois that reduces the trusted computing base required by web browsers. The trusted computing base is the subset of the software in which any exploitable error would lead to the system being potentially compromised. The problem with modern web browsers is that they have a huge trusted computing base and are integrated tightly into the actual operating system, and thus provide a convenient environment for malicious attackers to gain access to computer systems. By having a reduced trusted computing base, IBOS offers increased security with respect to modern web browsers.

IBOS is a combination of web browser and operating system. The trusted computing base is reduced by utilizing a microkernel and exposing browser-level abstractions at the lowest software layer. This approach allows for removal of almost all traditional OS components and services from the trusted computing base by directly mapping those browser abstractions to hardware abstractions. The resulting design turns out to be flexible enough to enable browser security policies while supporting traditional applications. In IBOS, for instance, device drivers, network protocol implementation, the storage stack, and window management software, among other system services, are outside of the trusted computing base. They all run on top of the trusted kernel of IBOS, which can enforce security policies. Also, the overhead added

to the browsing experience is small. See [101] for more details on IBOS.

Web-based applications (web apps) and the browser itself have become quite popular targets for attacks on computer systems. The vulnerabilities in web apps are ever increasing, so isolation of the web apps is highly desirable. Vulnerabilities in the actual web browsers are not as common as web app vulnerabilities, but occur often enough to be troubling. Further vulnerabilities are possible in the operating system, its services, and libraries.

Not all attacks are created equal. Attacks at the top of the software stack will only give the attacker access to the browser's current vulnerable web app. Further down the stack, attacks on the browser would give the attacker access to all web apps, their data, and system resources the browser can access. At the bottom of that stack, attacks on the operating system itself can be the most devastating, as the attacker can gain full control of the system. Vulnerabilities higher in the stack turn out to be more common, but are less damaging. Attacks lower in the stack have a much higher threat potential, and that is what IBOS is trying to address.

IBOS is designed to compartmentalize all the different processes as much as possible, and all communication is being forcibly routed through the trusted kernel. The IBOS kernel decides, based on the policies, which communication between processes is allowed, and thus possible. As will be seen in Section 6.2, the communication between different web page instances, network processes, the network card, the display memory, and the central kernel is modeled. In Section 6.3, safety properties referring to the immutability of the security policy are automatically obtained with help of the InvA tool. The formal analysis in this chapter is done under the *assumption of a correct underlying microkernel*. See [95] for a discussion about the features of the L4Ka::Pistachio microkernel and the fully verified seL4 microkernel.

6.1.1 IBOS Architecture

Figure 6.1, borrowed from [95], depicts a simplified view of IBOS architecture, in which the hardware is at the bottom of the stack, and the IBOS kernel and part of the trusted computing base (TCB) are on top of that. Everything on top of the kernel is not part of the TCB. Specifically, all web apps, network processes and the network interface card (NIC) driver do not need to be trusted. The following are observations about three important components of the IBOS architecture:

The IBOS kernel. The IBOS kernel builds upon the L4Ka microkernel

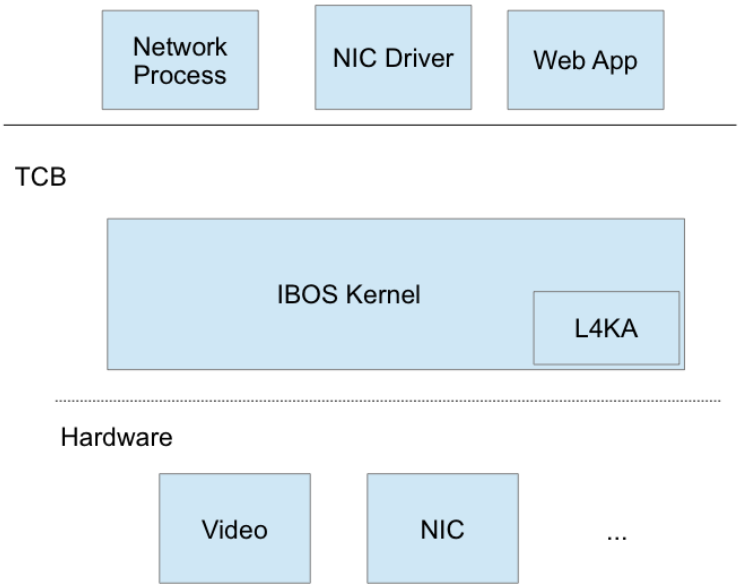

Figure 6.1: IBOS Architecture.

and is the central component of the IBOS web browser. It takes care of traditional OS tasks, e.g., process creation and application memory management. Message passing is based on the L4Ka::Pistachio message passing implementation, forcing all messages through the kernel, and specifically allows the checking of the security policies.

Network process. The network process is responsible for HTTP requests. It transforms HTTP data into a TCP stream and in turn into a series of Ethernet frames which are passed to the NIC driver.

Web apps. A new web app is created for each individual page visit of the user; specifically, whenever a link is clicked or a new URL is entered into the address bar. A web app sends out the HTTP request to the network process, parses HTML and runs JavaScript and renders web content to a tab. Each web app is labeled with the origin of the HTTP request used at creation.

A key property of the IBOS browser is that *all communication*, i.e., all messages sent or received, *get transmitted through the* IBOS *kernel*. This is because the message passing is implemented as system calls, which of course go to the microkernel operating system, which is tightly integrated with the IBOS kernel. The following are two key goals of IBOS:

- Security decisions happen at the lowest possible level: small TCB.

- Enough browser states and events are exposed, so as to allow for security policy checking; this makes IBOS flexible to allow new browser security policies.

See [101] for all IBOS goals and more detail.

6.2 Formal Modeling Methodology

The verification of some IBOS properties in this chapter uses a new Maude specification derived from the Maude specification of IBOS previously presented in [95, 96]. The new specification was obtained by source code inspection and from clarifications by R. Sasse when needed. It preserves reachability and at the same time satisfies the conditions required by the techniques in Chapter 4 for proving safety properties, and thus is amenable to mechanical analysis in the InvA tool. The documented source code of the new specification is available from `http://camilorocha.info/thesis`.

The Maude specification models the *architecture* of IBOS and includes: (i) the kernel, (ii) general message passing, (iii) web apps, (iv) network processes, and (v) network interface card access. The main component, the kernel, includes the policy checking mechanism for messages, an address bar, the content currently displayed on the screen, etc. The user interface (UI) is also part of the kernel.

All messages are forced to go through the kernel and they are thus subject to the policies it wants to enforce. This is already a design decision in IBOS, which the browser enforces, and it is reflected in the formal model in the way messages are passed. Each process can only directly send messages to the kernel, and the message will include the actual final destination in some way; but only the kernel is able to send messages to any of the processes. In the model, this is ensured by having two one-way pipes for messages for each process and the kernel, i.e., one incoming and one outgoing pipe. Thus, the kernel is the only connecting point and the policy checking is easily centralized.

As mentioned earlier, the Maude specification of IBOS presented in this section is amenable to mechanical verification in the InvA system, in contrast to the original specification [95]. This new version of the specification is a topmost rewrite theory which uses equality enrichments instead of built-in equality predicates. Rewrite rules for checking the security policy enforced by the system are preferred over auxiliary function symbols equationally defined, and the list datatype modeling the communication pipes does not use

any associativity axioms, since they are problematic for unification purposes. It is important to note that the new specification preserves reachability with respect to the original specification, but some clarifications are in order:

- The function symbol removed from the original specification in [95], appeared in the right-hand side of the rewrite rule that enforced the security policy of the browser. Such an auxiliary function symbol was originally defined by a collection of 10 equations that have been upgraded to equivalent topmost rewrite rules in the newer specification. The observation here is that one of the equations used Maude's `owise` programming feature for specifying that a message to the kernel should be dropped whenever none of the other 9 equations applied, i.e., whenever the security policy does not allow the kernel to handle the message. In the new specification a message to the kernel can be handled by the kernel if the security policy allows it or be dropped, even if the security policy allows the kernel to handle such a message.

- In the case of pipes, all but one rewrite rule in the original specification used the common "first-rest" matching for syntactic lists, thus making the use of any associativity axioms unnecessary in these cases. The remaining rewrite rule relied on the associativity axiom for nondeterministically choosing an element in a nonempty pipe. This rule is replaced by two rewrite rules in the new specification: one for moving the first element in a nonempty pipe to the end of the pipe, and another for choosing the top element in a nonempty pipe. Some of these changes are detailed in Section 6.2.1.

It can be argued that the success in proving safety properties for the Alternate Bit Protocol in Chapter 4 is a good indication that a formal modeling approach, as the one taken here, can provide assurance about a design if safety properties can be proved for the formal model. However, the formal modeling process is done completely by hand. This raises two issues: one being that a counterexample found in the model might not actually be a counterexample in the original specification. That is checkable and no actual false positive counterexamples have been identified. The second issue is that the formal model is an abstraction of the actual browser, as well as a translation of its code. Therefore, all security guarantees based on the this model are given with respect to the *design* and cannot guarantee the total absence of bugs introduced in the browser implementation.

The verification task in this chapter is different from the one in Chapter 4, in the sense that the IBOS specification is a novel and more challenging

benchmark, even though the verification techniques in both cases are the same. Also, this chapter is different from Chapter 4 in [95] in that the safety properties here are mechanically obtained and do not rely on boundedness arguments making the reachable state space finite, among other things.

6.2.1 IBOS Architecture Modeling

This section points out key properties and gives a general flavor of the model.

Figure 6.2, borrowed from [95], depicts the IBOS model state. At the top level, the state space is represented by the top sort `Sys`, which is made up of configurations of objects, all wrapped by curly brackets:

```
op {_} : Configuration -> Sys [ctor] .
```

Objects are formed by an object identifier, a type, and a set of attributes. Each network process, web app, and the kernel is modeled as a single object. In Figure 6.2, all objects outside the kernel are shown as rectangles. Pipes are a special kind of object connecting the objects at their left and right ends. Other than that, arrows show connectivity. The ellipses inside the kernel contain relevant pieces of the kernel, that are not objects themselves. There will be multiple copies of most objects, except for the NIC, display and web app manager. The uniqueness of the kernel process is verified in Section 6.3.2.

The Kernel and Message Passing.

All messages in the state are passed as system calls, where the browser-specific part of the message is encapsulated in the system call. First, the message part specific to the browser has the following format, called the `payload` of the encapsulating system call:

```
op payload : Oid Oid MsgType MsgVal Label
                              typed untyped -> Payload [ctor] .
```

The arguments of payload are the sender (as `Oid`), the receiver (as `Oid`), the message type (as `MsgType`), some auxiliary message info (as `MsgVal`), an argument commonly containing the URL that is requested or sent (as `Label`), and two more arguments (`typed` and `untyped`) that could transport more data (and which are ignored here). The sort `Oid` is that of object or process identifiers. Each web app, network process, etc., has an `Oid`. The correct sender `Oid` is enforced by the kernel, as it knows which process sent the system call encapsulating this `payload`.

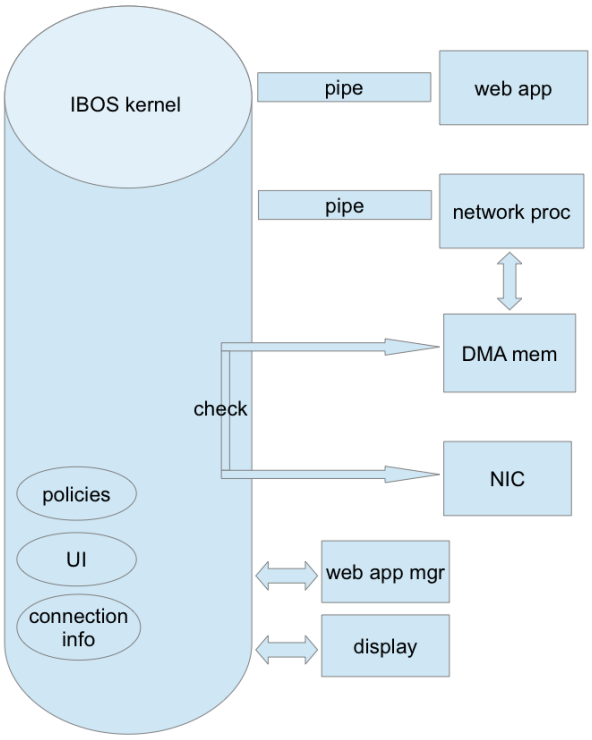

Figure 6.2: IBOS Model State.

The actual message is built using the `payload` and system call type:

```
op msg : SyscallType Payload -> Message [ctor] .
op OPOS-SYSCALL-FD-SEND-MESSAGE : -> SyscallType .
```

where `OPOS-SYSCALL-FD-SEND-MESSAGE` is the most commonly used type of system call for sending browser messages.

To model the fact that the kernel knows which process actually sent a message (as a system call) and to make sure that in the model no two processes can send messages directly to each other, but are forced to send messages via the kernel, the model defines one pipe object per process (using the same `Oid` as the associated process), which contains two one-way pipes, going to the kernel from the process and going to the process from the kernel:

```
op pipe       : -> Cid [ctor] .
op fromKernel : MessageList -> Attribute [ctor] .
op toKernel   : MessageList -> Attribute [ctor] .
```

For instance, a pipe object for the process with object identifier `1050`, which currently holds no message going either way, is represented as follows:

```
< 1050 : pipe | fromKernel(mt), toKernel(mt) >
```

Also consider the following message that is going to be sent by process `1050`:

```
msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
    payload(1050, 256, MSG-FETCH-URL, 0,
            l(http,dom("test"),port(81)),
            mtTyped, mtUntyped))
```

This message comes from web app `1050` and is addressed to network process `256`. It requests an URL (`MSG-FETCH-URL`) from domain `http://test:81`. In order to send the message, this is appended to the list of messages in `toKernel` in the pipe object.

The kernel, with object identifier `id(1)`, is only handling one thing at a time, which is stored in `handledCurrently`. Once the current instruction has been dealt with, any of the currently incoming messages can become the next message to be executed. The following rule enforces the policy checking when two process are allowed to exchange messages of a given type:

```
rl [kernelReceivesOPMessage-pa1] :
   { < id(1) : kernel |
         handledCurrently(none),
         msgPolicy(policy(ID:ProcId,ID2:ProcId,M:MsgType),
                   PS:PolicySet),
         Att:AttributeSet >
     < ID:ProcId : pipe |
         toKernel(msg(ST:SyscallType,
                      payload(ID1:ProcId, ID2:ProcId, M:MsgType,
                      V:MsgVal, L:Label, T:typed, U:untyped)),
                  ML:MessageList),
         Att2:AttributeSet >
       Cnf:Configuration }
=> { < id(1) : kernel |
          handledCurrently(
            msg(ST:SyscallType,
                payload(ID:ProcId, ID2:ProcId, M:MsgType,
                        V:MsgVal, L:Label, T:typed, U:untyped))),
          msgPolicy(policy(ID:ProcId,ID2:ProcId,M:MsgType),
                    PS:PolicySet),
          Att:AttributeSet >
       < ID : pipe |
           toKernel(ML:MessageList) ,
           Att2:AttributeSet >
       Cnf:Configuration } .
```

The policy checking is enforced by the matching condition on attributes `msgPolicy` of the kernel and `payload` in attribute `toKernel` of the communicating process pipe. More precisely, the property being checked here is that the message policy in `msgPolicy` allows the process `ID:ProcId` to send messages of type `M:MsgType` to process `ID2:ProcId`. Then the message is passed on to the kernel to become the next message to be executed. Note that the sender identifier `ID1:ProcId` of the incoming message is changed

to the actual sender identifier `ID:ProcId`, which is the process identifier of the pipe (and thus the associated process).

For the network processes, identifiers 256 through 1023 are being used, as does IBOS. The attribute names of a network process are:

```
op returnTo : ProcId -> Attribute [ctor] .
op in       : LabelList -> Attribute [ctor] .
op out      : LabelList -> Attribute [ctor] .
```

Attribute `returnTo` stores the process identifier of the web app that this network process will return data to, while attributes `in` and `out` hold the lists (or queues) of labels representing URLs that the network process will ask data from and has received data from already. As a simplification, a given URL label is used to represent the data from that URL instead of using the actual HTML code from the URL.

Process identifiers 1024 through 1055 are used for web apps. Their attributes names are:

```
op rendered : Label -> Attribute [ctor] .
op URL      : Label -> Attribute [ctor] .
op loading  : Nat -> Attribute [ctor] .
```

The label inside `rendered` is the URL for which the web app has put the data on the screen, provided it is the active web app. The label inside `URL` is the location where this web app wants to load data from. Attribute `loading` is just a binary flag indicating whether the web app has already sent a request to load data. Initially, the `rendered` attribute for a new web app will be empty, and `loading` is `0`, meaning that it has not yet started to load. The following topmost rule sends the message to start loading:

```
crl [fetch] :
    { < ID:ProcId : proc |
           rendered(L1:Label), URL(L2:Label), loading(0),
           Att:AttributeSet >
      < ID:ProcId : pipe |
           toKernel(ML:MessageList), Att2:AttributeSet >
      Cnf:Configuration }
 => { < ID:ProcId : proc |
           rendered(L1:Label), URL(L2:Label), loading(1),
           Att:AttributeSet >
      < ID:ProcId : pipe |
           toKernel(ML:MessageList ;
                    (msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
                         payload(ID:ProcId, id(4), MSG-FETCH-URL,
                                 0, L2:Label, mtTyped, mtUntyped)),
                     mt)),
           Att2:AttributeSet >
      Cnf:Configuration }
```

```
if isWebapp(ID:ProcId) .
```

The message is sent to the network process manager, which is identified with `id(4)`, for fetching the data from URL `L2` and the `loading` attribute changes to `1`. On return of the requested data, `L2` will be the new content of attribute `rendered`.

The hardware pieces of Figure 6.2, video card, NIC, etc., are not modeled in any detail. Only the NIC is modeled, and it receives target URLs from the memory set aside for this purpose through the kernel, and then, after a potential delay, returns the representation of the resulting data.

6.3 Address Bar Correctness and Some Auxiliary Invariants

The analysis to follow is based on the formal model explained in Section 6.2. An important property for a web browser is the trustworthiness of user interface elements. This is crucial, for instance, to counter spoofing attacks. Particularly, the address bar needs to be trustworthy, so that the user always knows which site is currently being visited: it is important to know whether the currently visited site is really the banking web site or is instead a phishing site, where risk is imminent. It is simple for malicious attackers to create phishing web sites that are indistinguishable on their looks from the real web sites. A user should be able to trust the address bar and be guaranteed that address bar spoofing attacks will not succeed. The goal of this section is to stress test the capabilities and limits of the InvA tool when trying to prove that address bar spoofing attacks are not possible in IBOS.

6.3.1 Formal Specification of the Property and Limitations

Address bar correctness in the browsing system model means that the content of the displayed page is always from the address displayed in the address bar. In the IBOS model, the kernel keeps track of the address bar by means of the data stored in the `displayedTopBar` attribute. The source of the content being displayed is stored in the display process abstraction, which has the `displayedContent` attribute to store this information. The content of both fields needs to be the same at all times, except when there currently is no content in one of the two fields. The empty content is modeled by the `about-blank` label.

The property being checked is modeled by the state predicate `inv`, where the kernel process has identifier `id(1)` and the display process abstraction has identifier `id(15)`:

```
op inv : Sys -> [Bool] .

eq inv( { < id(1) : kernel |
              displayedTopBar(about-blank), Att1:AttributeSet >
          < id(15) : proc |
              displayedContent(L1:Label), Att2:AttributeSet >
          Cnf:Configuration } )
  = true .
eq inv( { < id(1) : kernel |
              displayedTopBar(L1:Label), Att1:AttributeSet >
          < id(15) : proc |
              displayedContent(about-blank), Att2:AttributeSet >
          Cnf:Configuration } )
  = true .
ceq inv( { < id(1) : kernel |
              displayedTopBar(L1:Label), Att1:AttributeSet >
          < id(15) : proc |
              displayedContent(L2:Label), Att2:AttributeSet >
          Cnf:Configuration
        } )
    = L1:Label ~ L2:Label
 if L1:Label ~ about-blank = false
 /\ L2:Label ~ about-blank = false .
```

The first equation in the definition of `inv` defines the case where the kernel's displayed top bar is empty. Similarly, the second equation defines the case where the display's displayed content is empty. The last case, as defined by the third equation, is the most interesting one. In this third case, the invariant is defined to hold if the two aforementioned contents coincide whenever they both are nonempty. As a special remark, note that the operator `_~_` in the third equation is an equality enrichment for sort `Label` (see Section 2.3).

To motivate the property of address bar correctness, note that the address bar and the content as stored in the display process, are both stateless objects as they have no memory, but only what is currently stored.

Both the address bar and the display content are only changed due to the current web app interacting with the kernel when created or when the tab is switched to it. These two separate behaviors are modeled in the system by the rules `[new-url]`, `[tab-change]`, and `[change-display]`. The following is the rewrite rule `[change-display]` as modeled in IBOS:

```
crl [change-display] :
    { < id(15) : proc |
          activeWebapp(P:ProcId), displayedContent(L1:Label),
          Att1:AttributeSet >
```

```
     < P:ProcId : proc |
         rendered(L2:Label), Att2:AttributeSet >
     Cnf:Configuration }
 => { < id(15) : proc |
         activeWebapp(P:ProcId), displayedContent(L2:Label),
         Att1:AttributeSet >
     < P:ProcId : proc |
         rendered(L2:Label), Att2:AttributeSet >
     Cnf:Configuration }
 if L1:Label ~ L2:Label = false
 /\ isWebapp(P:ProcId) .
```

In summary, attribute `displayedContent` in the kernel process is changed by this rule to the content of attribute `rendered` in the active web app, whenever the two contents are different. This intuitively means that a content is displayed once the requested URL has been fetched by the system.

The goal is to obtain an automatic proof of the invariant:

$$\texttt{IBOS} \Vdash \texttt{init} \Rightarrow \texttt{inv}.$$

The following verification commands can be given to the InvA tool to check the above-mentioned invariant, as indicated by the inference rule Inv in Section 4.4 and its implementation in the InvA tool:

```
(analyze init(S:Sys) implies inv(S:Sys) in IBOS-PREDS .)

(analyze-stable inv(S:Sys) in IBOS-PREDS KERNEL .)
```

Before showing the output given by the tool, it is assumed that module `IBOS-PREDS` contains the state predicates and their corresponding auxiliary functions, and module `KERNEL` contains the specification of IBOS. State predicate `init` defines the initial state of the system.

When issuing the above commands, the InvA tool generates and tries to discharge more that 18000 proof obligations in less than 3 minutes:

```
Checking IBOS-PREDS ||- init(S:Sys) => inv(S:Sys) ...
rewrites: 28502 in 18ms cpu (18ms real) (1500342 rewrites/second)
Proof obligations generated:  1
Proof obligations discharged: 1
Success!

Checking IBOS-PREDS ||- inv(S:Sys) => O inv(S:Sys) ...
rewrites: 15810839 in 179525ms cpu (179557ms real)
  (88070 rewrites/second)
Proof obligations generated:  18120
Proof obligations discharged: 18072
The following proof obligations need to be discharged:
1808. from inv-2 & webapp-change-display : pending
inv({ < #8:ProcId : proc | #10:AttributeSet,rendered(#9:Label)>
```

```
      < id(1): kernel | displayedTopBar(#6:Label)>
      < id(15): proc | activeWebapp(#8:ProcId),
          displayedContent(#9:Label)>})
 = true
if about-blank ~ #6:Label = false
/\ about-blank ~ #7:Label = false
/\ #6:Label = #7:Label
/\ #7:Label ~ #9:Label = false
/\ isWebapp(#8:ProcId) = true .

...
```

There are 48 proof obligations returned to the user. That is, less than 0.2% of the verification task is left for the user. Upon inspection of the InvA's output, in this case all proof obligations shown to the user are generated from the rewrite rule `[change-display]`. This situation softens the burden for the user in the quest for a proof or a counterexample. A key observation about this rule, is that it updates the value of attribute `displayedContent` in the display process, regardless of the contents of attribute `displayedTopBar` in the kernel process. Therefore, a pattern is observed in which there is no inductive assumption on the relation of these two values that the inference system can use to discharge the pending proof obligations.

To illustrate the process, consider the proof obligation identified by label `1808` and shown above. It is perfectly possible to have a situation in where labels `#6` and `#7` represent the (fictitious but trusted) web site `mylifesavings.com`, and where label `#9` represents the (fictitious but distrusted) web site `danger.com`. Assuming that `#8` is a web app identifier, it is straightforward to see that an actual attack could happen and that the proof obligation in this case is false. In conclusion, state predicate `inv` is *not an inductive invariant* of IBOS in the sense of the inference system in Chapter 4. Therefore, a stronger property is required to imply the correctness of the address bar.

In order to come up with a strengthening of the invariant, it is key to understand the internal behavior of the system prior to changing the display. To illustrate a specific situation, consider the following flow of events from the moment when a new web app is created to display a given web page, say `L`, to the moment when the actual web page `L` is displayed in the content area of the browser:

1. Initially, the new web app is created with a new web app identifier `P`, attribute `URL` with content `L`, and attribute `rendered` with content `about-blank`. It becomes the active web app, which is represented in the system by updating the value of attribute `activeWebapp` to `P` in

the kernel. Also, attribute `displayedTopBar` in the kernel is updated to `L` and the new web app is associated to `L` in the kernel storage for web app connections `weblabels`. Attribute `displayedContent` in the display process is updated to `about-blank`.

2. Eventually, web app `P` will start loading, which is modeled by rewrite rule `[fetch]`, previously introduced in this section. This is achieved by sending a `MSG-FETCH-URL` message to the network processes requesting to fetch `L`. This message is placed in the process's pipe to later be checked by the kernel, which enforces the security policy on every message in the model. In the meantime, the new process waits for the requested data.

3. The URL fetch request from `P` for the resource `L` will eventually be checked by the kernel. The security policy enforced by the kernel will trigger one of three possible actions in the system:

   - If the security policy *does not allow communication* between web apps and network processes, then the message is dropped and ignored. In this case, the new web app will never get an answer and the empty content will remain displayed, which is fine for the purpose of proving the address bar correct.
   - If the security policy *allows communication* between web apps and network processes, then there are two cases: (i) if the resource `L` was previously requested by another web app, then the cached value in the network process connection storage `networklabels` attribute is returned (this is done by placing a message in the input pipe of the requesting web app with the cached value for `L`); (ii) if the resource has not been previously requested, then a memory process and an associated pipe are created. If the security policy allows it, the new memory process will communicate with the NIC driver process and ultimately will obtain the requested resource. As a net effect of the overall transaction, the network process connection storage `networklabels` attribute in the kernel will be updated with the cached value, and the message with this value will be forwarded to the web app initially created as a response.

   Presumably, the returned value, cached or not, will correspond to `L`.

4. In the event of receiving a `MSG-RETURN-URL` from a network process in

its input pipe with URL `L1`, attribute `rendered` is updated to `L1` and this process stops waiting for the requested resource. Since presumably the fetched value coincides with the requested one, then it must be the case that `L` and `L1` are the same. Therefore, at this point attributes `URL` and `rendered` in the newly created web app are the same.

5. Finally, the content stored in the display process `displayedContent` is updated to `L` and the safety property of interest is true at any point of execution. This is because if attribute `displayedContent` in the display process has initially the empty content, then it will eventually be updated to `L`, the value initially stored in attribute `displayedTopBar` of the kernel when `P` was created.

Note that there is a lot going on behind the scenes and important assumptions are being made. Basically, each item in the list above is part of a strengthening for `inv`, and thus needs to be formally verified. In the case study experiments, the goal of proving the address bar correct was temporarily abandoned because mechanically proving some of these strengthenings turned out to be quite challenging. The difficulty was mainly due to the large number of proof obligations that could not be automatically discharged by the tool, even though the tool *did* discard many other ones. At first glance, the tool was able to discharge more that 95% of tens of thousands of proof obligations it generated.

However, there are specific results in the verification task of `inv` that are promising in the light of the overall mechanical proof effort. In sections 6.3.2 and 6.3.3 the focus is on documenting automatic proofs obtained with InvA of two auxiliary invariants that are stepping stones in a future mechanical proof for the correctness of the address bar. Section 6.3.4 presents a discussion about current limitations of the InvA tool that were identified during the experiments with this case study.

6.3.2 Kernel Uniqueness

The kernel process plays a central role in the design of IBOS. This process is responsible for enforcing the security policy in the entire browsing system and does the bookkeeping for network process in the connection storage `networklabels`, among other things. Uniqueness of the kernel is a trivial property that the system should satisfy, but it needs to be formally proved. One first reason is that almost all interesting properties in IBOS have to do with message passing that is supervised by the kernel. Therefore a kernel

must exist and be unique. A second reason is that having this property can help in obtaining a simpler property specifications for IBOS because then this requirement does not need to be explicitly stated.

Kernel uniqueness (and existence) in the IBOS system is modeled by state predicate `unique-kernel` as follows:

```
 op unique-kernel : Sys -> Bool .
ceq unique-kernel( { Cnf } )
    = false
 if no-kernel(Cnf) .
 eq unique-kernel( { < P:ProcId : kernel | Att:AttributeSet >
                     Cnf:Configuration } )
  = no-kernel(Cnf:Configuration) .
 eq unique-kernel( { < P1:ProcId : kernel | Att1:AttributeSet >
                     < P2:ProcId : kernel | Att2:AttributeSet >
                     Cnf:Configuration } )
  = false .

 op no-kernel : Configuration -> Bool .
 eq no-kernel(none)
  = true .
 eq no-kernel( < P:ProcId : C:Cid | Att:AttributeSet >
               Cnf:Configuration )
  = not(C:Cid ~ kernel) and no-kernel(Cnf:Configuration) .
```

The equational definition of `no-kernel` fully defines the predicate and considers three cases: the first when no kernel exists, the second when exactly one kernel exists, and a third in which at least two kernels exist. Obviously, the second case is the only one in which the predicate should hold. On input `Cnf:Configuration`, auxiliary function `no-kernel` holds if and only if there is no kernel process in `Cnf`. Also note that `no-kernel` uses the Boolean equality enrichment defined for the sort `Cid` of class identifiers. As an additional remark, observe that both `unique-kernel` and `no-kernel` preserve ground confluence, termination, and sort-decreasingness.

The goal with this predicate is to prove:

$$\texttt{IBOS} \Vdash \texttt{init} \Rightarrow \Box\texttt{unique-kernel}.$$

The outcome of the verification task in the InvA tool for this invariant is shown below:

```
rewrites: 33699 in 84ms cpu (84ms real) (401154 rewrites/second)
Checking IBOS-PREDS ||- init(S:Sys) => unique-kernel(S:Sys) ...
Proof obligations generated:  1
Proof obligations discharged: 1

rewrites: 944276 in 8724ms cpu (8762ms real) (108232 rewrites/second)
```

```
Checking
  IBOS-PREDS ||- unique-kernel(S:Sys) => O unique-kernel(S:Sys) ...
Proof obligations generated:  2592
Proof obligations discharged: 2592
Success!
```

A total of 2593 proof obligations are generated and discharged by the InvA tool, thus providing an automatic mechanical proof of the invariance of `unique-kernel` for IBOS. As a side margin note, from the experience with this specification, in some cases it is convenient to prove uniqueness and existence for other processes in the model besides the kernel. In such a case, the approach presented here can be used as a template for obtaining easy automatic proofs.

6.3.3 Immutability of the Security Policy

One important goal is to ensure that the IBOS kernel upholds the security policy, even if one or more of the subsystems have been compromised. Note that the security policy is trusted, for instance, at any point of execution in the list of events in Section 6.3.1. This property turns out to be important, not only as a stepping stone for proving other invariants, but also in its own right.

For instance, consider a threat model, initially proposed in [101], in which an attacker controls a web site and can feed arbitrary data to the browser. Indeed, it can be assumed that this malicious data or application can compromise one or more of the components in the system, such as the drivers and processes. Once the attacker gains control of these components, arbitrary instructions can be executed as a result of the attack. The aim, then, is to maintain the integrity and confidentiality of the data in the browser. More specifically, the goal is to guarantee that if a user opens a web page in a trusted web server, then this user can interact with this page securely, even if everything on the client system outside the TCB has been compromised.

The layers upon which IBOS is built are trusted. These layers include the underlying hardware. IBOS enforces security decisions based on its security policy, so it is important to establish safety properties about the integrity of its security policy. More specifically, even if the system is subjected to a threat model such as the one just described, one would like to have the guarantee that the security policy can not be altered by the attacker. Of course, compromising any of the underlying layers trusted by the kernel could compromise the security of IBOS.

The immutability of the security policy for IBOS is modeled by state predicate `immutable-policy` as follows:

```
op immutable-policy : Sys PolicySet -> [Bool] .
eq immutable-policy(
   { < id(1) : kernel | msgPolicy(PS:PolicySet), Att:AttributeSet >
     Cnf:Configuration }, PS:PolicySet)
 = true .
```

State predicate `immutable-policy` is parametric on security policy sets. It holds if the policy set used by the kernel coincides with the given policy set. Note that this predicate is only defined for the positive case, which is fine for mechanical analysis in InvA, as explained in Chapter 4. Also, in order for the analysis to be consistent, it is assumed that there is exactly one kernel in the configuration, which has been established as an inductive invariant of IBOS in Section 6.3.2.

The initial policy set is specified by means of constant `init-policy`:

```
op init-policy : -> PolicySet .
eq init-policy
 = ( ... ) .
```

The goal with this predicate is to prove:

$$\texttt{IBOS} \Vdash \texttt{init} \Rightarrow \Box\texttt{immutable-policy}(\texttt{init-policy}).$$

By using the proof system for proving safety properties implemented in the InvA tool (see Chapter 4 for details), the following results are obtained when analyzing the invariance of `immutable-policy` with respect to the security policy initially defined for IBOS:

```
rewrites: 28501 in 72ms cpu (72ms real) (395819 rewrites/second)
Checking
  IBOS-PREDS ||- init(S:Sys)
                 => immutable-policy(S:Sys,init-policy) ...
Proof obligations generated:  1
Proof obligations discharged: 1
Success!

rewrites: 444761 in 3260ms cpu (3258ms real)
  (136421 rewrites/second)
Checking
  IBOS-PREDS ||- immutable-policy(S:Sys,PS:PolicySet)
                 => O immutable-policy(S:Sys,PS:PolicySet) ...
Proof obligations generated:  2088
Proof obligations discharged: 2088
Success!
```

The InvA tool generates more than 2000 proof obligations and mechani-

cally proofs that the security policy is immutable in any reachable state by automatically discharging all these proof obligations in less than 4 seconds. In particular, this result asserts that the security policy, once defined will never change even if everything outside the TCB has been compromised. It is important to note that the inductive stability proof just shown has been obtained for *any* policy set `PS:PolicySet`, and not only for the particular case of `init-policy`. More precisely, the stability proof is actually a more general proof of the form:

$$\texttt{IBOS} \Vdash \texttt{immutable-policy(PS:PolicySet)} \Rightarrow \Box\texttt{immutable-policy(PS:PolicySet)}.$$

This ultimately means that *any* security policy defined as part of IBOS's initial state is immutable.

6.3.4 Discussion on Some Limits of InvA

The IBOS case study is the most challenging specification analyzed in the InvA tool so far. The new IBOS specification used in this proof effort comprises more than 1150 lines of source code, 47 equations, and 26 rewrite rules, which results in thousands of proof obligations for most verification tasks performed with help of the InvA tool. As a point of comparison, the ABP specification studied in Chapter 4 comprises 208 lines of code, 20 equations, and has half the rewrite rules of IBOS. The largest number of proof obligations for a particular invariant in the ABP case study was about 100. As previously seen, InvA was successful in discharging thousands of proof obligations and automatically proving some invariants. However, the IBOS case study has unveiled limitations of the InvA tool which, from the perspective of stress testing the limits of the tool, is also a positive experience.

The following paragraphs identify and summarize some limitations of the InvA tool. They also propose possible solutions that should make the mechanical verification of invariants in InvA effective for large specifications, including the correctness of the address bar for IBOS:

User Support. There should be better management of proof obligations, specially when analyzing large specifications: it is very complicated, time consuming, and error-prone to analyze a list of almost 400 proof obligations! For instance, consider the following header output by the tool:

```
Checking
  IBOS-PREDS ||- good-webapps(S:Sys) => O good-webapps(S:Sys) ...
```

```
Proof obligations generated:  2000
Proof obligations discharged: 1608
The following proof obligations need to be discharged:
...
```

In the worst case, the user needs to go through *all* of the 392 proof obligations in order to understand what the situation is, and whether or not the state predicate `good-webapps` is a promising inductive invariant. In this example, it was easy to find a proof obligation indicating that `good-webapps` is *not* an inductive invariant, but this may not be the case in general.

New Heuristics. There is a need for improving the proof heuristics used by the tool. As explained in Chapter 4, a series of heuristics are employed by the InvA for discharging proof obligations. However, it should be possible to improve some of them and implement some new ones. For example, the InvA tool implements some basic heuristic for checking unsatisfiability of numeric conditions modulo SMT. This could perhaps be combined with equational narrowing, which is already available in Maude. This should increase the number of proof obligations automatically discharged by the tool, and thus lessen the proof effort of the user.

Inductive Techniques. There is also the need for improving the techniques available to the user in tools such as the ITP. These could help in obtaining easy interactive proofs in many cases where the proof obligations cannot be discharged automatically. As it was the case with the IBOS specification, many data types in the state are actually sets or multisets. Inductive techniques such as cover-set induction modulo AC should be investigated, implemented, and offered to the user. The current ITP version supports cover-set induction [52] but for the moment *not* modulo AC.

6.4 Related Work and Concluding Remarks

Formal verification of the IBOS system in Maude has been done recently by R. Sasse [95, 96]. The correctness of the address bar was obtained with bounded model checking in Maude and reduction techniques that resulted in a finitary and tractable state space that could then be inspected with Maude's `search` command. One difference between R. Sasse's approach and the one followed here is the ultimate goal of the verification task: in the latter, the goal was to obtain mechanical proofs without user interaction and, at the same time, stress test the InvA tool implementation.

Important earlier work on Internet Explorer was done in Maude [21], where graphical user interface security has been addressed and previously unknown attack types (for each of which an actual malicious web page could succeed in an attack) were uncovered. Formal modeling has been done before for the OP2 [47] browser. IBOS is based on some of the ideas of OP2 but takes them further by, among other things, reducing the trusted computing base.

The IBOS case study is the most challenging specification analyzed in the InvA tool so far, which was successful in discharging thousands of proof obligations and automatically proving some invariants. However, the IBOS case study has unveiled limitations of the InvA tool which, from the perspective of stress testing the limits of the tool, is also a positive experience. This chapter identifies and summarizes these limitations, and also proposes possible solutions that should make the mechanical verification of invariants in InvA effective for large specifications, including the correctness of the address bar for IBOS.

CHAPTER 7

REACHABILITY ANALYSIS WITH CONSTRAINED BUILT-INS

Rewriting logic theories have been successfully used for modeling and executing concurrent systems with rich data structures and very general forms of transitions. It would not be misleading to say that one of the most attractive features for day-to-day verification purposes of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ in Maude, is the implementation of general on-the-fly algorithms for model checking properties of the initial reachability model $\mathcal{T}_{\mathcal{R}}$. These techniques include Maude's `search` command and Maude's LTL model checker. However, the use of such tools often assumes that the set of terms representing the state search space of $\mathcal{R}$ are indeed ground. Could it be possible to have a more general rewrite relation that operates on terms that are not necessarily ground and is at the same time amenable to verification with the same Maude tools?

A direct but naive answer to the above question is yes: just do what is often done in first order logic by means of the *theorem of constants* and treat the variables in the state terms as constants by enlarging the signature of $\mathcal{R}$ with new constants. However, this approach has a major drawback: the "blind" codification of non-ground terms as ground terms with new constants will in general not be complete with respect to the initial semantics of $\mathcal{T}_{\mathcal{R}}$. The reason for this is that the matching performed when computing the rewrite relation for the extended rewrite theory uses constructor patterns and therefore can miss substitutions on terms that contain the new constants. An alternative solution would be to use the narrowing-based approach; but then the main objective of using Maude's convenient on-the-fly verification mechanisms could directly be impacted since new narrowing-based model checking tools (such as, for example, the Maude-NPA tool for

model checking cryptographic protocol [37]) are needed. In practice, the main limitation of the narrowing-based methods is that their symbolic decision procedures are often difficult to find and, in some cases when available, special purpose decision procedures such as those implemented by SMT solvers can be more efficient.

The goal of this chapter is to introduce the notion of a *constrained rewrite theory*, an extension of rewriting logic theories in which the rewrite rules can be used to *rewrite* terms with *constrained built-ins*. That is, terms involving user-definable data structures but whose only variables range over decidable domains. The main advantage of these theories is that, under some mild syntactic conditions and the availability of an oracle for the constraints (such as an SMT solver), they can be directly encoded in Maude and thus induce a *symbolic rewrite* relation that is amenable, for example, to model checking verification using Maude's `search` command and LTL model checker.

This chapter is organized as follows. Section 7.1 presents the notion of constrained built-in term. Section 7.2 introduces the notion of constrained rewrite theory, symbolic atomic relation, and explains the relationship between the symbolic reachability semantics associated to the symbolic atomic relation on constrained terms and its initial reachability semantics. The soundness and completeness of the symbolic simulation with constrained built-ins, under appropriate assumptions, is presented in Section 7.3. Section 7.4 extends the symbolic atomic relation associated to a constrained rewrite theory to more general cases of rewriting such as asynchronous and parallel closures. Section 7.5 presents some related work. A case study on a symbolic rewriting logic semantics for PLEXIL based on these methods is presented in Chapter 9. The symbolic semantics complements the ground rewriting logic semantics of PLEXIL, presented in Chapter 8, with symbolic detection of reachability violations on input plans where the values of external variables can be left unspecified.

7.1 Terms with Constrained Built-ins

The notion of built-ins for the order-sorted equational theory $\mathcal{E} = (\Sigma, E)$ is modeled with a many-sorted equational theory $\mathcal{E}_\Lambda = (\Lambda, E_\Lambda)$ such that the inclusion $\mathcal{E}_\Lambda \subseteq \mathcal{E}$ is protecting. The intuition is that $\mathcal{T}_{\mathcal{E}_\Lambda}$ is the subalgebra of built-in terms of $\mathcal{T}_\mathcal{E}$. It is assumed that the built-in function symbols F_Λ of Λ are disjoint from any other symbols in Σ. Then $\Sigma = (S, \leq, F)$ can be decomposed into $\Lambda = (S_\Lambda, F_\Lambda)$ and $\Sigma' = (S, \leq, F \setminus F_\Lambda)$. The collection

$X_\Lambda \subseteq X$ denotes the S_Λ-indexed set of variables $X_\Lambda = \{X_s\}_{s \in S_\Lambda}$.

A *constraint* is a Boolean combination of Λ-equalities. Constraints are interpreted in a $\mathcal{E}_\Lambda$-algebra $\mathcal{M} = (M, \_\mathcal{M})$, with *domain* $M = \{M_s\}_{s \in S_\Lambda}$ and *interpretation function* $\_\mathcal{M} = \{\_\mathcal{M},s\}_{s \in S_\Lambda}$, satisfying $\mathcal{T}_{\mathcal{E}_\Lambda} \simeq \mathcal{M}$. The collection of constraints is denoted by $\Lambda(X_\Lambda)$. The distinction between the isomorphic algebras $\mathcal{T}_{\mathcal{E}_\Lambda}$ and $\mathcal{M}$ is stressed to emphasize the fact that the approach presented here can use an 'oracle' for solving constraints (using term rewriting, SMT solving, theorem proving, etc.). Note that $\mathcal{E}$ is a *protecting extension* of $\mathcal{E}_\Lambda$ and therefore $\mathcal{T}_\mathcal{E}|_\Lambda \simeq \mathcal{T}_{\mathcal{E}_\Lambda} \simeq \mathcal{M}$.

A *constrained term* is a pair $\langle t\,;\varphi_t\rangle$ in $T_\Sigma(X_\Lambda) \times \Lambda(X_\Lambda)$ and its *denotation* $\langle t\,;\varphi_t\rangle_\mathcal{M}$ is the set:

$$\langle t\,;\varphi_t\rangle_\mathcal{M} = \{t' \mid (\exists \sigma : X_\Lambda \longrightarrow T_\Lambda)\; t' = t\sigma \;\wedge\; \mathcal{M} \models \varphi_t\sigma\}. \tag{7.1}$$

The domain of σ in the definition above ranges over all built-in variables X_Λ and consequently $\langle t\,;\varphi_t\rangle_\mathcal{M} \subseteq T_{\Sigma,s}$ for any sort $s \in S$ and term $t \in T_\Sigma(X_\Lambda)_s$, even if $vars(t) \not\subseteq vars(\varphi_t)$. Given $s \in S$, a constrained term $\langle t\,;\varphi_t\rangle$ is said to be an *s-constrained term* if and only if $t \in T_\Sigma(X_\Lambda)_s$. Σ-terms not containing any built-in variables will not be considered as first components of constrained terms.

7.2 Atomic Relations for Constrained Terms

This section introduces a symbolic term rewrite relation on terms with constrained built-in subterms and assumes the notation introduced in Section 7.1.

The symbolic term rewrite relation on terms with constrained built-ins is called the *symbolic atomic relation* and is defined by a *constrained rewrite theory*. The symbolic atomic relation is intended to be a building block for more general symbolic relations, such as the asynchronous and/or parallel closures that are commonly used for specifying the semantics of programming languages.

A constrained rewrite theory assumes the choice of a top sort $\mathfrak{s}$ in one of the connected components of Σ.

Definition 15 (Constrained Rewrite Theory)**.** *A* $\mathcal{M}$-constrained Σ-rule *(or* constrained rule*) is a triple* $l \to r \,[\![\varphi]\!]$ *where:*

a. l *and* r *are terms in* $T_\Sigma(X)_\mathfrak{s}$ *with* $l \notin X$ *and* $vars(r) \setminus vars(l, \varphi) \subseteq X_\Lambda$*,*

b. $\varphi \in \Lambda(X_\Lambda)$ *is satisfiable in* $\mathcal{M}$*, and*

c. l is linear and if $t \in T_\Lambda(X_\Lambda)$ is a proper subterm of l, then $t \in X_\Lambda$.

Terms l and r are called, respectively, the lefthand *and* righthand *side, and φ the* constraint *of the constrained rule. An $\mathcal{M}$-*constrained rewrite theory *(or* constrained rewrite theory*) is a tuple (Σ, E, R), where R is a finite collection of $\mathcal{M}$-constrained Σ-rules.*

A constrained rule can contain extra built-in variables in its righthand side and excludes the case of an unsatisfiable constraint. The lefthand side of a constrained rule is linear and does not refer to the operator structure of the built-ins, not even to constants. These restrictions are key to the completeness result proved in this section. On the other hand, they are not overly restrictive for many practical applications. For example, consider a rule

$$a(x_1 + (x_2 - x_1)) \to b(x_2) \, [\![\varphi]\!] \,,$$

where a and b are non built-in unary function symbols in $F \setminus F_\Lambda$, $+$ and $-$ are built-in function symbols in F_Λ, x_1 and x_2 are built-in variables in X_Λ, and φ is constraint in $\Lambda(X_\Lambda)$. This rule does *not* conform to Condition (c) in Definition 15 because x_1 occurs more than once in the lefthand side and $x_1 + (x_2 - x_1)$ is a proper built-in subterm of the lefthand side that is not a built-in variable. However, this rule can be transformed by applying the technique of *variable abstraction* into the following constrained rule with an extra built-in variable y_1 in X_Λ:

$$a(y_1) \to b(x_2) \, [\![\varphi \wedge y_1 = x_1 + (x_2 - x_1)]\!] \,.$$

The relation induced by a constrained rewrite theory on $\mathfrak{s}$-constrained terms is called the *symbolic atomic relation* and is introduced in Definition 16.

Definition 16 (Symbolic Atomic Relation)**.** *Let $\mathcal{R} = (\Sigma, E, R)$ be a constrained rewrite theory such that the variables in E and R are in the finite set $Y \subseteq X$. The* symbolic atomic relation $\rightsquigarrow_{\mathcal{R}}$ *induced by $\mathcal{R}$ on constrained terms denotes the set of pairs in $(T_\Sigma(X_\Lambda \setminus Y)_\mathfrak{s} \times \Lambda(X_\Lambda \setminus Y))^2$ such that $\langle t \,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}} \langle u \,;\varphi_u\rangle$ if and only if there are $l \to r \, [\![\varphi]\!] \in R$ and $\theta : X \longrightarrow T_\Sigma(X)$ satisfying*

a. $t =_E l\theta$ and $u =_E r\theta$,

b. $\mathcal{M} \models (\varphi_u \equiv \varphi_t \wedge \varphi\theta)$, and

c. φ_u *is satisfiable in* $\mathcal{M}$.

Intuitively, the symbolic atomic relation $\rightsquigarrow_{\mathcal{R}}$ on constrained terms is defined as the topmost rewrite relation induced by R modulo E on $T_\Sigma(X_\Lambda)$ with extra bookkeeping of constraints. Note that φ_u in $\langle t\,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}} \langle u\,;\varphi_u\rangle$, when witnessed by $l \to r\,[\![\varphi]\!]$ and θ, is *semantically equivalent* to $\varphi_t \wedge \varphi\theta$ in $\mathcal{M}$, in contrast to being *syntactically* equal. This freedom allows for simplification of constraints if desired. Also, such a constraint φ_u is satisfiable in $\mathcal{M}$, implying that φ_t and $\varphi\theta$ are both satisfiable in $\mathcal{M}$, and therefore $\langle t\,;\varphi_t\rangle_{\mathcal{M}} \neq \varnothing \neq \langle u\,;\varphi_u\rangle_{\mathcal{M}}$. The assumption that the variables occurring in E and R are disjoint from the ones occurring in the constrained terms is a well-known technical requirement, so that matching a term $t \in T_\Sigma(X_\Lambda)$ with the lefthand of a rule does not wrongly capture variables. Without this requirement in Definition 16, it would be problematic to have a variable in $\mathit{vars}(t) \setminus \mathit{vars}(l)$ that occurs in φ.

The binary relation induced by a constrained rewrite theory on $T_{\Sigma,\mathfrak{s}}$ is called the *atomic relation*.

Definition 17 (Atomic Relation)**.** *Let* $\mathcal{R} = (\Sigma, E, R)$ *be a constrained rewrite theory. The* atomic relation $\to_{\mathcal{R}}$ *induced by* $\mathcal{R}$ *denotes the set of pairs in* $T^2_{\Sigma,\mathfrak{s}}$ *such that* $t' \to_{\mathcal{R}} u'$ *if and only if there are* $l \to r\,[\![\varphi]\!] \in R$ *and ground substitution* $\sigma : X \longrightarrow T_\Sigma$ *satisfying*

a. $t' =_E l\sigma$ *and* $u' =_E r\sigma$*, and*

b. $\mathcal{M} \models \varphi\sigma$.

The atomic relation $\to_{\mathcal{R}}$ is the topmost rewrite relation induced by R modulo E on $T_{\Sigma,\mathfrak{s}}$. This relation is defined even when a rule in R has extra variables in its righthand side: such extra variables are assumed to be arbitrarily instantiated. Also, note that non built-in variables can occur in l, but the constraint $\varphi\sigma$ is a *variable-free sentence* in $\Lambda(X_\Lambda)$, so that the expression $\mathcal{M} \models \varphi\sigma$ is well defined.

7.3 Soundness and Completeness

This section assumes the notation introduced in Section 7.1.

The first question to ask is whether the symbolic atomic relation $\rightsquigarrow_{\mathcal{R}}$ soundly and completely simulates the atomic relation $\to_{\mathcal{R}}$. It is important to highlight that the notions of soundness and completeness discussed here

are *relative* to the model $\mathcal{M}$ and should *not* be understood as the traditional notions in first-order logic. Soundness can be proved directly from the definitions. For completeness the idea is that, assuming the admissibility conditions for $\mathcal{E}$, matching modulo axioms for $T_\Sigma(X_\Lambda)$ is enough for characterizing the complete set of ground unifiers (i.e., solutions) for a subclass of $T_\Sigma(X_\Lambda)$.

The soundness of $\rightsquigarrow_\mathcal{R}$ w.r.t. $\rightarrow_\mathcal{R}$ is stated and proved in Lemma 4. Intuitively, soundness means that a pair $\langle t\,;\varphi_t\rangle \rightsquigarrow_\mathcal{R} \langle u\,;\varphi_u\rangle$ is a symbolic underapproximation of all pairs such that $t' \rightarrow_\mathcal{R} u'$ with $t' \in \langle t\,;\varphi_t\rangle_\mathcal{M}$ and $u' \in \langle u\,;\varphi_u\rangle_\mathcal{M}$.

Lemma 4 (Soundness)**.** *Let $\mathcal{R} = (\Sigma, E, R)$ be a constrained rewrite theory, $t, u \in T_\Sigma(X_\Lambda)_\mathbb{s}$, and $\varphi_t, \varphi_u \in \Lambda(X_\Lambda)$. If $\langle t\,;\varphi_t\rangle \rightsquigarrow_\mathcal{R} \langle u\,;\varphi_u\rangle$, then $t\rho \rightarrow_\mathcal{R} u\rho$ for all $\rho : X_\Lambda \longrightarrow T_\Lambda$ satisfying $\mathcal{M} \models \varphi_u\rho$.*

Proof. Let $\rho : X_\Lambda \longrightarrow T_\Lambda$ be such that $\mathcal{M} \models \varphi_u\rho$. The goal is to show that $t\rho \rightarrow_\mathcal{R} u\rho$, i.e., that there exists a ground substitution $\sigma : X_\Lambda \longrightarrow T_\Lambda$ such that $t\rho =_E l\sigma$, $u\rho =_E r\sigma$, and $\mathcal{M} \models \varphi\sigma$. Let $l \rightarrow r\,[\![\varphi]\!]$ and $\theta : X_\Lambda \longrightarrow T_\Lambda(X_\Lambda)$ witness $\langle t\,;\varphi_t\rangle \rightsquigarrow_\mathcal{R} \langle u\,;\varphi_u\rangle$, with $vars(t,\varphi_t) \cap vars(l,r,\varphi) = \varnothing$. Then $t =_E l\theta$, $u =_E r\theta$, $\mathcal{M} \models (\varphi_u \equiv \varphi_t \wedge \varphi\theta)$, and φ_u is satisfiable in $\mathcal{M}$. Without loss of generality assume that $\theta|_{vars(t,\varphi_t)}$ is the identity and let $\sigma = \theta\rho$. Then note that $t\rho =_E (l\theta)\rho = l(\theta\rho) = l\sigma$ and $u\rho =_E (r\theta)\rho = r(\theta\rho) = r\sigma$. Moreover, $\mathcal{M} \models (\varphi_u \equiv \varphi_t \wedge \varphi\theta)$ and $\mathcal{M} \models \varphi_u\rho$ imply $\mathcal{M} \models \varphi\theta\rho$, i.e., $\mathcal{M} \models \varphi\sigma$. Therefore, $t\rho \rightarrow_\mathcal{R} u\rho$, as desired. □

Under the admissibility assumptions, $\mathcal{R}$ has a disjoint union $E \uplus B$ of equations, with B a collection of structural axioms for which there exists a decidable and finitary matching algorithm modulo B, and E a set of orientable sort-decreasing, operationally terminating, confluent rewrite rules modulo B. Moreover, if the axioms B are such that they are regular and collapse-free for built-in sorts, then matching modulo B can capture the ground solutions of equalities modulo B for a subset of $T_\Sigma(X_\Lambda)$. More specifically, the ground instances of the set of B-matching substitutions of a term $t \in T_\Sigma(X_\Lambda)_\mathbb{s}$ to a term $l \in T_\Sigma(X_\Lambda)_\mathbb{s}$ that satisfies Condition (c) in Definition 15 exactly characterize the set $GU_B(t = l)$. This observation is made precise and proved in Lemma 5.

Lemma 5 (Lifting Lemma)**.** *Let $t \in T_\Sigma(X_\Lambda)_\mathbb{s}$ and $l \in T_\Sigma(X)_\mathbb{s}$. If $vars(t) \cap vars(l) = \varnothing$, B is regular and collapse-free for all sorts in S_Λ, and l satisfies Condition (c) in Definition 15, then*

$$t \ll_B l \qquad \Longleftrightarrow \qquad GU_B(t = l) \neq \varnothing.$$

Proof. ($\Longrightarrow$) If $t \ll_B l$, then $t =_B l\theta'$ for some $\theta' : X \longrightarrow T_\Sigma(X)$. Let $\theta : X \longrightarrow T_\Sigma(X)$ be defined by $\theta = \theta'|_{vars(l)} \cup id|_{X \setminus vars(l)}$. Since $\theta|_{vars(t)} = id|_{vars(t)}$, it follows that $t\theta =_B t =_B l\theta$. Hence, $t\theta\sigma =_B l\theta\sigma$ for any $\sigma : X \longrightarrow T_\Lambda$, and then $GU_B(t = l) \neq \varnothing$. ($\Longleftarrow$) Without loss of generality assume $l \in T_\Sigma(X_\Lambda)$, since otherwise, because $vars(t) \subseteq X_\Lambda$, any variable $x \in vars(l) \cap (X \setminus X_\Lambda)$ can be substituted by a ground subterm of t obtaining a term $\bar{l}$ such that $GU_B(t = l) = \varnothing$ if and only if $GU_B(t = \bar{l}) = \varnothing$. Recall that Σ can be decomposed into Λ and $\Sigma' = (S, \leq, F \setminus F_\Lambda)$. If the axioms B are collapse-free for any sort in S_Λ, then $B = B_\Lambda \cup B'$ for some sets of Σ-equations B_Λ, with B' such that B_Λ operates on $T_\Lambda(X_\Lambda)$ and B' on $T_{\Sigma'}(X)$. Then $l \in T_{\Sigma'}(X_\Lambda)$, since it satisfies Condition (c) in Definition 15 and by the assumption above. Moreover, l can be viewed as the n-ary function $L(x_1, \ldots, x_n)$ with $\mathfrak{s}$-context $L = \lambda x_1, \ldots, x_n.l$, where $\{x_1, \ldots, x_n\} = vars(l)$. Note that, by l linear, there is a 1-to-1 correspondence between the x_i's and the built-in subterms of l. Similarly, t can be viewed as the m-ary function $C(t_1, \ldots, t_m)$, with $\mathfrak{s}$-context $C = \lambda y_1, \ldots, y_m.c$ and $c \in T_{\Sigma'}(\{y_1, \ldots, y_m\})_\mathfrak{s}$. Without loss of generality assume $\{y_1, \ldots, y_m\} \cap \{x_1, \ldots, x_n\} = \varnothing$. If $\sigma \in GU_B(t = l)$, then

$$\begin{aligned} C(t_1\sigma, \ldots, t_m\sigma) &= C(t_1, \ldots, t_m)\sigma \\ &= t\sigma \\ &=_B l(x_1, \ldots, x_n)\sigma \\ &= l(x_1\sigma, \ldots, x_n\sigma). \end{aligned}$$

Moreover, $C(y_1, \ldots, y_m)$ and $l(x_1, \ldots, x_n)$ must be B'-equal up to renaming of variables. Since B' is regular, it follows that $m = n$. Then, there exists a permutation $\varphi : \{1, ..., n\} \longrightarrow \{1, ..., n\}$ satisfying

$$C(x_{\varphi(1)}, \ldots, x_{\varphi(n)}) =_{B'} l(x_1, \ldots, x_n) \text{ and } t_{\varphi(i)}\sigma =_{B_\Lambda} x_i\sigma \text{ for } 1 \leq i \leq n.$$

Let $\theta : X \longrightarrow T_\Lambda(X_\Lambda)$ be defined by $\theta(x_i) = t_{\varphi(i)}$ for $1 \leq i \leq n$, and be the identity id elsewhere. Note that θ is well-defined because Λ is many-sorted. Also note that $vars(t) \cap vars(l) = \varnothing$ imply

$$t = C(t_{\varphi(1)}, \ldots, t_{\varphi(n)}) =_{B'} l(t_{\varphi(1)}, \ldots, t_{\varphi(n)}) =_{B_\Lambda} l(x_1, \ldots, x_n)\theta.$$

Therefore $t \ll_B l$, as desired. □

The requirement of being collapse-free for built-in sorts is key in the proof of Lemma 5: it allows viewing B-matching for $T_\Sigma(X_\Lambda)$ as a modular combination of B_Λ-matching for $T_\Lambda(X_\Lambda)$ and B'-matching for $T_{\Sigma'}(X)$. This idea

was adopted and adapted from the development in [77]. Any combination of associativity, commutativity, and identity axioms is regular and has a decidable and finitary matching algorithm. However, only combinations of associativity and commutativity are collapse free for any sort since these axioms are sort-preserving. For identity axioms there is a special treatment as they are allowed in B as long as they are collapse-free for any built-in sort. The restriction on identity axioms collapsing non built-in terms into built-in terms can be handled by a theory transformation that eliminates the need for identity axioms, such as the one described in [32]. As a final remark, note that there is a finitary matching algorithm for associativity axioms in contrast to a nonexistent general and finitary unification algorithm. This means that the symbolic reachability method developed here based on the symbolic atomic relation can be applied to specifications with lists, queues, and stacks that often use associativity axioms, and that are outside the scope of general-purpose unification-based methods such as narrowing.

As a side margin note, observe that Lemma 5 uses the same motivation as Lemma 2 in Section 4.2.1. The main difference is that the former uses matching instead of unification for computing a complete set of solutions for ground equalities.

The completeness of $\rightsquigarrow_{\mathcal{R}}$ w.r.t. $\rightarrow_{\mathcal{R}}$ is stated and proved in Lemma 6. Intuitively, completeness states that $\rightsquigarrow_{\mathcal{R}}$ is a symbolic overaproximation of $\rightarrow_{\mathcal{R}}$. In Lemma 6 it is assumed that the lefthand side of each constrained rewrite rule in R is $\rightarrow_{E/B}$-irreducible. Such an assumption is easily met in practice because if l is a lefthand side in R that is $\rightarrow_{E/B}$-reducible, by confluence and operational termination it has an unique E/B-canonical form to which variable abstraction can be applied, thus obtaining an equivalent constrained rule.

Lemma 6 (Completeness)**.** *Let $\mathcal{R} = (\Sigma, E \cup B, R)$ be a constrained rewrite theory, $t \in T_\Sigma(X_\Lambda)_\mathfrak{s}$, $u' \in T_{\Sigma,\mathfrak{s}}$, and $\varphi_t \in \Lambda(X_\Lambda)$. Assume that $(\Sigma, E \cup B)$ is admissible, the axioms in B are regular and are collapse-free for any sort in S_Λ, and the lefthand side of the rules in R are in E-canonical form modulo B. For any $\rho : X_\Lambda \longrightarrow T_\Lambda$ such that $t\rho \in \langle t\,;\varphi_t\rangle_{\mathcal{M}}$ and $t\rho \rightarrow_{\mathcal{R}} u'$, there exist $u \in T_\Sigma(X_\Lambda)_\mathfrak{s}$ and $\varphi_u \in \Lambda(X_\Lambda)$ such that $\langle t\,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}} \langle u\,;\varphi_u\rangle$ and $u' \in \langle u\,;\varphi_u\rangle_{\mathcal{M}}$.*

Proof. Let ρ be as given, and let $l \rightarrow r\,[\![\varphi]\!] \in R$ and $\sigma : X_\Lambda \longrightarrow T_\Lambda$ witness $t\rho \rightarrow_{\mathcal{R}} u'$, i.e., $t\rho =_{E\cup B} l\sigma$, $u' =_{E\cup B} r\sigma$, and $\mathcal{M} \models \varphi\rho$, with $vars(t, \varphi_t) \cap vars(l, r, \varphi) = \varnothing$. The goal is to show the existence of $u \in T_\Sigma(X_\Lambda)_\mathfrak{s}$ and $\varphi_u \in \Lambda(X_\Lambda)$ s.t. (i) $\langle t\,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}} \langle u\,;\varphi_u\rangle$ and (ii) $u' \in \langle u\,;\varphi_u\rangle_{\mathcal{M}}$. Without

loss of generality assume $\rho = \sigma$. Under the admissibility conditions for $(\Sigma, E \cup B)$, the term $\bar{t} = t\downarrow_{E/B}$ is well-defined and satisfies $\bar{t} \in T_\Sigma(X_\Lambda)_\mathfrak{s}$ and $vars(\bar{t}) \subseteq vars(t)$. Then $\bar{t}\sigma =_{E\cup B} t\sigma =_{E\cup B} l\sigma$. Since l is in E/B-canonical form by assumption and $\bar{t}$ is in E/B-canonical form, it follows that $\bar{t}\sigma =_B l\sigma$. Therefore $GU_B(\bar{t} = l) \neq \varnothing$ and, with the given assumptions, Lemma 5 yields the existence of $\theta : X \longrightarrow T_\Sigma(X)$ such that $\bar{t} =_B l\theta$; a fortiori, $t =_{E\cup B} l\theta$. Without loss of generality assume $\theta|_{X\setminus vars(l)} = \sigma|_{X\setminus vars(l)}$. Take $u = r\theta$ and $\varphi_u = \varphi_t \wedge \varphi\theta$. First note that if $x \in vars(r)$, then $x \in vars(l)$ or $x \in X \setminus vars(l)$. If $x \in vars(l)$, then $\theta(x) \in T_\Sigma(X_\Lambda)$ because $\theta(x)$ is a subterm of $\bar{t} \in T_\Sigma(X_\Lambda)$; if $x \in X \setminus vars(l)$, then $\theta(x) = \sigma(x) \in T_\Sigma$. That is, $u = r\theta \in T_\Sigma(X_\Lambda)_\mathfrak{s}$. On the other hand, $t\sigma = t\rho \in \langle t\,;\varphi_t\rangle_\mathcal{M}$ by assumption and then $\mathcal{M} \models \varphi_t\sigma$. Also $t\sigma = t\rho =_{E\cup B}= l\sigma$ and $t =_{E\cup B} l\theta$, imply $\theta\sigma =_{E\cup B} \sigma$. Then $\mathcal{M} \models \varphi\theta\sigma$, because $\mathcal{M} \models \varphi\rho$ by assumption and $\rho = \sigma$. That is, $\mathcal{M} \models \varphi_t\sigma \wedge \varphi\theta\sigma$ and therefore $\mathcal{M} \models \varphi_u\sigma$ by definition of φ_u. Summarizing: $t =_{E\cup B} l\theta$, $u = r\theta$, $\varphi_u = \varphi_t \wedge \varphi\theta$, and φ_u is satisfiable in $\mathcal{M}$. Therefore (i) follows. For (ii) note that the already proven facts $u' =_{E\cup B} r\sigma$, $\sigma =_{E\cup B} \theta\sigma$, and $\mathcal{M} \models \varphi_u\sigma$ imply $u' =_{E\cup B} r\sigma \in \langle r\theta\,;\varphi_u\rangle_\mathcal{M}$ with witness σ. □

Under the admissibility assumptions for $\mathcal{E} = (\Sigma, E\cup B)$, each $t \in T_\Sigma(X_\Lambda)_\mathfrak{s}$ has an E/B-canonical form $\bar{t}$ with sort $\mathfrak{s}$ and is unique modulo B. Also, the canonical form $\bar{t}$ is such that $vars(\bar{t}) \subseteq vars(t)$.

Theorem 12 collects the soundness and completeness statements of $\rightsquigarrow_\mathcal{R}$ w.r.t. $\rightarrow_\mathcal{R}$.

Theorem 12. *Let $\mathcal{R} = (\Sigma, E \cup B, R)$ be a constrained rewrite theory, $t \in T_\Sigma(X_\Lambda)_\mathfrak{s}$, and $\varphi_t \in \Lambda(X_\Lambda)$.*

(Soundness) *If $u \in T_\Sigma(X_\Lambda)_\mathfrak{s}$, $\varphi_u \in \Lambda(X_\Lambda)$, and $\langle t\,;\varphi_t\rangle \rightsquigarrow_\mathcal{R} \langle u\,;\varphi_u\rangle$, then*

$$(\forall \rho : X_\Lambda \longrightarrow T_\Lambda) \quad \mathcal{M} \models \varphi_u\rho \implies t\rho \rightarrow_\mathcal{R} u\rho.$$

(Completeness) *If $(\Sigma, E\cup B)$ is admissible, B is regular and collapse-free for S_Λ, and the lefthand sides of the rules in R are in E/B-canonical form, and if $u' \in T_{\Sigma,\mathfrak{s}}$ and $\rho : X_\Lambda \longrightarrow T_\Lambda$ are such that $t\rho \rightarrow_\mathcal{R} u'$ and $t\rho \in \langle t\,;\varphi_t\rangle_\mathcal{M}$, then*

$$(\exists u \in T_\Sigma(X_\Lambda)_\mathfrak{s}, \varphi_u \in \Lambda(X_\Lambda)) \quad \langle t\,;\varphi_t\rangle \rightsquigarrow_\mathcal{R} \langle u\,;\varphi_u\rangle \ \wedge\ u' \in \langle u\,;\varphi_u\rangle.$$

Proof. By Lemma 4 and Lemma 6. □

It remains to be shown when Theorem 12 is effective, i.e., when $\rightsquigarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{R}}$ can be computed.

Lemma 7. *Let $\mathcal{R} = (\Sigma, E \cup B, R)$ be a constrained rewrite theory. Assume $(\Sigma, E \cup B)$ is admissible. Then:*

a. if there is a decision procedure for satisfiability of equalities and inequalities in $\mathcal{M}$, $\rightarrow_{\mathcal{R}}$ and $\rightsquigarrow_{\mathcal{R}}$ are computable;

b. if each constrained rule $l \rightarrow r \, [\![\varphi]\!] \in R$ is such that $vars(\varphi) \subseteq vars(l)$, $\rightarrow_{\mathcal{R}}$ is decidable.

The existence of unique E/B-canonical forms for terms in $T_\Sigma(X_\Lambda)_{\mathfrak{s}}$ is not enough for computing $\rightarrow_{\mathcal{R}}$ or $\rightsquigarrow_{\mathcal{R}}$, even when solving Boolean combinations of equalities with variables modulo B is decidable. The explanation for this is that membership in $\rightarrow_{\mathcal{R}}$ or $\rightsquigarrow_{\mathcal{R}}$ depends on inductive satisfiability of constraints, i.e., on satisfiability in the initial algebra $\mathcal{T}_{\Sigma/E}$ of constraints that can have variables. For example, it is easy to devise an admissible equational theory for specifying integer arithmetic, but it is well-known that non-linear integer arithmetic is undecidable.

7.4 Symbolic Closures

This section provides an account of symbolic relations induced by a symbolic atomic relation, such as the reflexive-transitive closure and the asynchronous closure among others. Soundness and completeness for each symbolic relation w.r.t. to its ground counterpart are corollaries of Theorem 12. In the development of this section, statements about soundness and completeness are to be understood as in the statement of Theorem 12, including the assumptions.

In the sequel it is assumed that the equational theories $\mathcal{E} = (\Sigma, E)$ and $\mathcal{E}_\Lambda = (\Lambda, E_\Lambda)$, and the $\mathcal{E}_\Lambda$-algebra $\mathcal{M}$ are as defined in Section 7.2. It is also assumed that the atomic relations induced by a constrained rewrite theory $\mathcal{R} = (\Sigma, E, R)$ are sort-decreasing. The symbolic atomic relation $\rightsquigarrow_{\mathcal{R}}$ is *sort-decreasing* if and only if for any $t, u \in T_\Sigma(X_\Lambda)_{\mathfrak{s}}$ and $\varphi_t, \varphi_u \in \Lambda(X_\Lambda)$ if $\langle t \,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}} \langle u \,;\varphi_u\rangle$ then $ls(t) \geq ls(u)$. The atomic relation $\rightarrow_{\mathcal{R}}$ is *sort-decreasing* if and only if for any $t', u' \in T_{\Sigma,\mathfrak{s}}$ if $t' \rightarrow_{\mathcal{R}} u'$ then $ls(t') \geq ls(u')$.

Let $\rightarrow$ be a binary relation on a given set. The *identity relation*, *n-fold composition*, and *reflexive-transitive closure* of $\rightarrow$ are defined as usual and denoted, respectively, by $(\rightarrow)^0$, $(\rightarrow)^n$, and $(\rightarrow)^*$ (or simply $\rightarrow^0$, $\rightarrow^n$, and $\rightarrow^*$). Note that $\rightarrow$ and $\rightarrow^1$ denote the same binary relation.

Corollaries 2 and 3 show, respectively, that the n-fold composition $\rightsquigarrow_{\mathcal{R}}^{n}$ and reflexive-transitive closure $\rightsquigarrow_{\mathcal{R}}^{*}$ of the symbolic atomic relation $\rightsquigarrow_{\mathcal{R}}$ are sound and complete w.r.t. to $\rightarrow_{\mathcal{R}}^{n}$ and $\rightarrow_{\mathcal{R}}^{*}$, respectively.

Corollary 2 (Fold Composition)**.** *For each $n \in \mathbb{N}$, $\rightsquigarrow_{\mathcal{R}}^{n}$ is sound and complete w.r.t. $\rightarrow_{\mathcal{R}}^{n}$.*

Proof. Let $t \in T_{\Sigma}(X)_{\mathfrak{s}}$ and $\varphi_t \in \Lambda(X_\Lambda)$.

(Soundness). Let $u \in T_{\Sigma}(X_\Lambda)_{\mathfrak{s}}$, $\varphi_u \in \Lambda(X_\Lambda)$, and $\rho : X_\Lambda \longrightarrow T_\Lambda$. The goal is to prove for any $n \in \mathbb{N}$ that if $\langle t\,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}}^{n} \langle u\,;\varphi_u\rangle$ and $\mathcal{M} \models \varphi_u\rho$, then $t\rho \rightarrow_{\mathcal{R}}^{n} u\rho$.

$n = 0:$ then $t = u$, $\varphi_t = \varphi_u$, and then $t\rho \rightarrow_{\mathcal{R}}^{0} t\rho$.

$n > 0:$ then there are $v \in T_{\Sigma}(X_\Lambda)_{\mathfrak{s}}$ and $\varphi_v \in \Lambda(X_\Lambda)$ such that

$$\langle t\,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}}^{n-1} \langle v\,;\varphi_v\rangle \rightsquigarrow_{\mathcal{R}} \langle u\,;\varphi_u\rangle.$$

Since $\mathcal{M} \models \varphi_u\rho$, by the soundness of $\rightsquigarrow_{\mathcal{R}}$ w.r.t. $\rightarrow_{\mathcal{R}}$ it follows that $v\rho \rightarrow_{\mathcal{R}} u\rho$. By definition of $\rightsquigarrow_{\mathcal{R}}$, φ_u implies φ_v in $\mathcal{M}$ and then $\mathcal{M} \models \varphi_v\rho$. By the induction hypothesis $t\rho \rightarrow_{\mathcal{R}}^{n-1} v\rho$. Therefore,

$$t\rho \rightarrow_{\mathcal{R}}^{n-1} v\rho \rightarrow_{\mathcal{R}}^{1} u\rho \qquad \text{, i.e.,} \qquad t\rho \rightarrow_{\mathcal{R}}^{n} u\rho.$$

(Completeness). Let $u' \in T_{\Sigma,\mathfrak{s}}$ and $\rho : X_\Lambda \longrightarrow T_\Lambda$ satisfy $t\rho \rightarrow_{\mathcal{R}}^{n} u'$ and $t\rho \in \langle t\,;\varphi_t\rangle$. For each $n \in \mathbb{N}$, the goal is to prove the existence of $u \in T_{\Sigma}(X_\Lambda)_{\mathfrak{s}}$ and $\varphi_u \in \Lambda(X_\Lambda)$ such that $\langle t\,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}}^{n} \langle u\,;\varphi_u\rangle$ and $u' \in \langle u\,;\varphi_u\rangle_{\mathcal{M}}$.

$n = 0:$ then $u = t$ and $\varphi_u = \varphi_t$ witness $\langle t\,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}}^{0} \langle u\,;\varphi_u\rangle = \langle t\,;\varphi_t\rangle$ and $u' = t\rho \in \langle t\,;\varphi_t\rangle_{\mathcal{M}} = \langle u\,;\varphi_u\rangle_{\mathcal{M}}$.

$n > 0:$ then there is $v' \in T_{\Sigma,\mathfrak{s}}$ such that $t\rho \rightarrow_{\mathcal{R}}^{n-1} v' \rightarrow_{\mathcal{R}} u'$. By the induction hypothesis, there are $v \in T_{\Sigma}(X_\Lambda)_{\mathfrak{s}}$ and $\varphi_v \in \Lambda(X_\Lambda)$ such that $\langle t\,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}}^{n-1} \langle v\,;\varphi_v\rangle$ and $v' \in \langle v\,;\varphi_v\rangle_{\mathcal{M}}$. Let $\sigma : X_\Lambda \longrightarrow T_\Lambda$ witness $v' \in \langle v\,;\varphi\rangle_{\mathcal{M}}$. Then $v' =_E v\sigma \rightarrow_{\mathcal{R}} t'$. By the completeness of $\rightsquigarrow_{\mathcal{R}}$ w.r.t. $\rightarrow_{\mathcal{R}}$ there are $u \in T_{\Sigma}(X_\Lambda)_{\mathfrak{s}}$ and $\varphi_u \in \Lambda(X_\Lambda)$ such that $\langle v\,;\varphi_v\rangle \rightsquigarrow_{\mathcal{R}} \langle u\,;\varphi_u\rangle$ and $u' \in \langle u\,;\varphi_u\rangle_{\mathcal{M}}$. Therefore $\langle t\,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}}^{n} \langle u\,;\varphi_u\rangle$ and $u' \in \langle u\,;\varphi_u\rangle_{\mathcal{M}}$, as desired.

□

Corollary 3 (Reflexive-Transitive Closure)**.** $\rightsquigarrow_{\mathcal{R}}^{*}$ *is sound and complete w.r.t.* $\rightarrow_{\mathcal{R}}^{*}$.

Proof. Follows by Corollary 2. □

The *symbolic asynchronous relation* $\overset{\triangle}{\rightsquigarrow}_{\mathcal{R}}$ is the binary relation on pairs in $T_\Sigma(X_\Lambda) \times \Lambda(X_\Lambda)$ defined by $\langle t\,;\varphi_t\rangle \overset{\triangle}{\rightsquigarrow}_{\mathcal{R}} \langle u\,;\varphi_u\rangle$ if and only if there is an $\mathfrak{s}$-context $C = \lambda x.c$ and terms $t_1, u_1 \in T_\Sigma(X_\Lambda)_\mathfrak{s}$ such that $t = C(t_1)$, $u = C(u_1)$, and $\langle t_1\,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}} \langle u_1\,;\varphi_u\rangle$. The *asynchronous relation* $\overset{\triangle}{\rightarrow}_{\mathcal{R}}$ is the binary relation on T_Σ defined by $t' \overset{\triangle}{\rightarrow}_{\mathcal{R}} u'$ if and only if there is an $\mathfrak{s}$-context $C = \lambda x.c$ and terms $t'_1, u'_1 \in T_{\Sigma,\mathfrak{s}}$ such that $t' = C(t'_1)$, $u' = C(u'_1)$, and $t'_1 \rightarrow_{\mathcal{R}} u'_1$. Note that the relations $\overset{\triangle}{\rightsquigarrow}_{\mathcal{R}}$ and $\overset{\triangle}{\rightarrow}_{\mathcal{R}}$ are well-defined because $\rightsquigarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{R}}$ are sort-decreasing.

Corollary 4 (Asynchronous Closure)**.** *The following statements hold:*

a. $\overset{\triangle}{\rightsquigarrow}_{\mathcal{R}}$ *is sound and complete w.r.t.* $\overset{\triangle}{\rightarrow}_{\mathcal{R}}$.

b. For each $n \in \mathbb{N}$, $(\overset{\triangle}{\rightsquigarrow}_{\mathcal{R}})^n$ *is sound and complete w.r.t.* $(\overset{\triangle}{\rightarrow}_{\mathcal{R}})^n$.

c. $(\overset{\triangle}{\rightsquigarrow}_{\mathcal{R}})^*$ *is sound and complete w.r.t.* $(\overset{\triangle}{\rightarrow}_{\mathcal{R}})^*$.

Proof. a. Let $t \in T_\Sigma(X_\Lambda)$ and $\varphi_t \in \Lambda(X_\Lambda)$.

(Soundness). Let $u \in T_\Sigma(X_\Lambda)$, $\varphi_u \in \Lambda(X_\Lambda)$, and $\rho : X_\Lambda \longrightarrow T_\Lambda$. The goal is to show that if $\langle t\,;\varphi_t\rangle \overset{\triangle}{\rightsquigarrow}_{\mathcal{R}} \langle u\,;\varphi_u\rangle$ and $\mathcal{M} \models \varphi_u\rho$, then $t\rho \overset{\triangle}{\rightarrow}_{\mathcal{R}} u\rho$. By definition of $\overset{\triangle}{\rightsquigarrow}_{\mathcal{R}}$, there is an $\mathfrak{s}$-context $C = \lambda x.c$ and terms $t_1, u_1 \in T_\Sigma(X_\Lambda)_\mathfrak{s}$ such that $t = C(t_1)$, $u = C(t_2)$, and $\langle t_1\,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}} \langle u_1\,;\varphi_u\rangle$. Then, $t_1\rho \rightarrow_{\mathcal{R}} u_1\rho$ by $\mathcal{M} \models \varphi_u\rho$ and the soundness of $\rightsquigarrow_{\mathcal{R}}$ w.r.t. $\rightarrow_{\mathcal{R}}$. Let $\theta : X_\Lambda \longrightarrow T_\Lambda(X_\Lambda)$ be defined by $\theta(x) = x$ and $\theta(x') = \sigma(x')$ otherwise, and define $C' = \lambda x.c'$, where $c' = c\theta$. Observe that C' is an $\mathfrak{s}$-context and that $\mathit{vars}(c') = \{x\}$. Moreover $t\rho = C'(t_1)\rho = C'(t_1\rho)$ and $u\rho = C'(u_1)\rho = C'(u_1\rho)$ imply $t\rho \overset{\triangle}{\rightarrow}_{\mathcal{R}} u\rho$ because $t_1\rho \rightarrow_{\mathcal{R}} u_1\rho$.

(Completeness). Let $u' \in T_\Sigma$ and $\rho : X_\Lambda \longrightarrow T_\Lambda$ satisfy $t\rho \overset{\triangle}{\rightarrow}_{\mathcal{R}} u'$ and $t\rho \in \langle t\,;\varphi_t\rangle$. The goal is to prove the existence of $u \in T_\Sigma(X_\Lambda)$ and $\varphi_u \in \Lambda(X_\Lambda)$ such that $\langle t\,;\varphi_t\rangle \overset{\triangle}{\rightsquigarrow}_{\mathcal{R}} \langle u\,;\varphi_u\rangle$ and $u' \in \langle u\,;\varphi_u\rangle_{\mathcal{M}}$. By definition of $\overset{\triangle}{\rightarrow}_{\mathcal{R}}$, there are $\mathfrak{s}$-context $C = \lambda x.c$ and terms $t_1^\rho, u_1^\rho \in T_{\Sigma,\mathfrak{s}}$ such that $t\rho = C(t_1^\rho)$, $u' = C(u_1^\rho)$, and $t_1^\rho \rightarrow_{\mathcal{R}} u_1^\rho$. Without loss of generality assume $\mathit{vars}(c) = \{x\}$. Note then that there is $t_1 \in T_\Sigma(X_\Lambda)_\mathfrak{s}$ subterm of t such that $t_1^\rho = t_1\rho$. Note also that $t\rho \in \langle t\,;\varphi_t\rangle_{\mathcal{M}}$ implies $t_1\rho \in \langle t_1\,;\varphi_t\rangle_{\mathcal{M}}$. Then, by completeness of $\rightsquigarrow_{\mathcal{R}}$ w.r.t. $\rightarrow_{\mathcal{R}}$, there are $u_1 \in T_\Sigma(X_\Lambda)_\mathfrak{s}$ and $\varphi_u \in \Lambda(X_\Lambda)$ satisfying

$\langle t_1 \,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}} \langle u_1 \,;\varphi_u\rangle$ and $u_1^\rho \in \langle u_1 \,;\varphi_u\rangle_{\mathcal{M}}$. In particular, $u_1^\rho =_E u_1\rho$ (see the proof of Theorem 12). Let $u = C(u_1)$, which is well-defined because $\rightsquigarrow_{\mathcal{R}}$ is sort-decreasing. Then it follows that $t = C(t_1) \overset{\triangle}{\rightsquigarrow}_{\mathcal{R}} C(u_1) = u$, and $u' = C(u_1^\rho) = C(u_1\rho) = C(u_1)\rho = u\rho \in \langle u \,;\varphi_u\rangle$, as desired.

b. Follows by part (a) and by mimicking the proof of Corollary 2.

c. Follows by part (b).

□

Note that Corollary 4 is of special interest for declarative specification and programming languages such as Maude [23] that implement equational and non-equational deduction via the asynchronous closure of the respective term rewriting relations. Corollary 4 also makes explicit a significant difference between $\overset{\triangle}{\rightsquigarrow}_{\mathcal{R}}$ and the asynchronous closure of *any* reasonable unification-based symbolic atomic relation. In the former, a deduction step within a context would require the propagation of the witnessing unifier to the entire context. However, this is not the case for $\rightsquigarrow_{\mathcal{R}}$, because there is no variable renaming or specialization on the subject term when it is matched with the lefthand side of a constrained rule.

The *symbolic parallel relation* $\overset{\shortparallel}{\rightsquigarrow}_{\mathcal{R}}$ is the binary relation on $T_\Sigma(X_\Lambda) \times \Lambda(X_\Lambda)$ defined by $\langle t \,;\varphi_t\rangle \overset{\shortparallel}{\rightsquigarrow}_{\mathcal{R}} \langle u \,;\varphi_u\rangle$ if and only if there is an $\mathfrak{s}$-context $C = \lambda x_1, \ldots, x_m.c$, terms $t_1, \ldots, t_m$ and $u_1, \ldots, u_m$ in $T_\Sigma(X_\Lambda)_\mathfrak{s}$, and constraints $\varphi_1, \ldots, \varphi_m$ in $\Lambda(X_\Lambda)$ satisfying $t = C(t_1, \ldots, t_m)$, $u = C(u_1, \ldots, u_m)$, $\langle t_i \,;\varphi_t\rangle \rightsquigarrow_{\mathcal{R}} \langle u_i \,;\varphi_i\rangle$ for $1 \leq i \leq m$, $\mathcal{M} \models (\varphi_u \equiv \wedge_{i=1}^m \varphi_i)$, and φ_u is satisfiable in $\mathcal{M}$. The *parallel relation* $\overset{\shortparallel}{\rightarrow}_{\mathcal{R}}$ is the binary relation on T_Σ defined by $t' \overset{\shortparallel}{\rightarrow}_{\mathcal{R}} u'$ if and only if there is an $\mathfrak{s}$-context $C = \lambda x_1, \ldots, x_m.c$ and terms $t'_1, \ldots, t'_m$ and $u'_1, \ldots, u'_m$ in $T_{\Sigma,\mathfrak{s}}$ satisfying $t' = C(t'_1, \ldots, t'_m)$, $u' = C(u'_1, \ldots, u'_m)$, and $t'_i \rightarrow_{\mathcal{R}} u'_i$ for $1 \leq i \leq m$. Note that the relations $\overset{\shortparallel}{\rightsquigarrow}_{\mathcal{R}}$ and $\overset{\shortparallel}{\rightarrow}_{\mathcal{R}}$ are well-defined because $\rightsquigarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{R}}$ are sort-decreasing.

Corollary 5 (Parallel Closure)**.** *The following statements hold:*

a. $\overset{\shortparallel}{\rightsquigarrow}_{\mathcal{R}}$ *is sound and complete w.r.t.* $\overset{\shortparallel}{\rightarrow}_{\mathcal{R}}$*.*

b. For each $n \in \mathbb{N}$, $(\overset{\shortparallel}{\rightsquigarrow}_{\mathcal{R}})^n$ *is sound and complete w.r.t.* $(\overset{\shortparallel}{\rightarrow}_{\mathcal{R}})^n$*.*

c. $(\overset{\shortparallel}{\rightsquigarrow}_{\mathcal{R}})^*$ *is sound and complete w.r.t.* $(\overset{\shortparallel}{\rightarrow}_{\mathcal{R}})^*$*.*

Proof.

a. Follows by an easy generalization of the proof for Corollary 5 part (a).

b. Follows by part (a) and by mimicking the proof of Corollary 2.

c. Follows by part (b).

□

The symbolic parallel closure generalizes the asynchronous closure of a symbolic atomic relation to several redexes and, like the latter, does not need to propagate the witnessing substitutions to the context. On the other hand, the symbolic parallel relations in Corollary 5 are useful in obtaining formal and executable semantics of synchronous languages by term rewriting. Such an approach is taken, for instance in [91] and [92], where an equational serialization procedure is used to simulate the parallel closure of an atomic relation via an asynchronous term rewriting relation.

7.5 Related Work and Concluding Remarks

The idea of combining term-rewriting techniques and constraint data structures is a hot topic of research nowadays, specially since the raise of modern theorem provers with highly efficient decision procedures in the form of SMT solvers. The overall aim is to advance applicability of rewriting techniques in verification by focusing on rewriting with constraints expressed in some logic that has an efficient decision procedure (see [79] for an overview).

M. Ayala-Rincón [3] investigates, in the setting of many-sorted equational logic, the expressiveness of conditional equational systems whose conditions may use built-in predicates. This class of equational theories is important because the combination of equational and built-in premises yield a type of clauses which is more expressive than purely conditional equations. Rewriting notions like confluence, termination, and critical pairs are investigated. In the context of rewriting logic, S. Falke and D. Kapur [40] have studied the problem of termination with constrained built-ins. In particular, they have extended the dependency pairs framework to handle termination of equational specifications with semantic data structures and evaluation strategies (as explained in Chapter 3) in the Maude language. The same authors have used the idea of combining rewriting induction and linear arithmetic over constrained terms [41]. Their aim is to obtain equational decision procedures that can handle semantic data types represented by the constrained built-ins.

CHAPTER 8

A REWRITING LOGIC SEMANTICS FOR PLEXIL

Synchronous languages were introduced in the 1980s to program *reactive systems*, i.e., systems whose behavior is determined by their continuous reaction to the environment where they are deployed. They are often used to program embedded applications and automatic control software. The family of synchronous languages is characterized by the *synchronous hypothesis*, which states that a reactive system is arbitrarily fast and able to react immediately in no time to stimuli from the external environment. One of the main consequences of the synchronous hypothesis is that components running in parallel are perfectly synchronized and cannot arbitrarily interleave. The implementation of a synchronous language usually requires the simulation of the synchronous semantics within an asynchronous computation model. This simulation must ensure the validity of the synchronous hypothesis in the target asynchronous model.

The *Plan Execution Interchange Language* (PLEXIL) [39] is a synchronous language developed by NASA to support autonomous spacecraft operations. Space mission operations require flexible, efficient and reliable plan execution. The computer system on board the spacecraft that executes plans is called the *executive* and is a safety-critical component of the space mission. The *Universal Executive* (UE) [103] is an open source PLEXIL executive developed by NASA. PLEXIL and the UE have been used on mid-size applications such as robotic rovers and a prototype of a Mars drill. It has also been used to demonstrate automation for the International Space Station.

Given the safety-critical nature of spacecraft operations, PLEXIL's operational semantics has been formally defined [30] and several properties of the language, such as determinism and compositionality, have been mechanically verified [29] in the Prototype Verification System (PVS) [82]. The formal

small-step semantics is defined using a compositional layer of five reduction relations on sets of *nodes*. These nodes are the building blocks of a PLEXIL plan and represent the hierarchical decomposition of tasks. From an operational point of view, PLEXIL is more complex, and more high-level, than general-purpose synchronous languages such as Esterel [11] or Lustre [19]. PLEXIL is designed specifically for flexible and reliable command execution in autonomous applications.

This chapter presents a rewriting logic semantics of PLEXIL specified in Maude. This semantics complements the small-step structural operational semantics written in PVS but, in contrast to the PVS higher-order logic specification, the rewriting logic semantics of PLEXIL is *executable* and therefore provides an interpreter for the language. This interpreter is used at NASA as an oracle and semantic standard for validating the implementation of PLEXIL executives, such as the UE, and as a language design infrastructure for designers of the language to study new features or possible variants of the language. Additionally, by using a graphical interface [84], PLEXIL developers are able to more easily and conveniently exploit the formal analysis tools provided by Maude to verify properties of actual plans.

A fruitful collaboration with the PLEXIL development team at NASA Ames has been established by using the rewriting logic semantics of PLEXIL to validate the intended semantics of the language against a wide variety of plan examples. Two problematic issues about PLEXIL's original semantics were discovered with the help of the rewriting logic semantics of PLEXIL presented in this chapter. The first was found at the level of the atomic relation, for which undesired interleaving semantics were introduced in some computations. The second was found at the level of the micro relation, for which spurious infinite loops were present in some computations. Solutions to both issues have been provided and validated using the rewriting logic semantics, and have been adopted in the latest version of the PLEXIL semantics.

This chapter is organized as follows. Section 8.1 presents an overview of the PLEXIL language. Section 8.2 introduces `PLX`, the rewriting logic semantics of PLEXIL, and illustrates its main design features. Section 8.3 summarizes some of the main contributions made possible by the rewriting logic semantics `PLX`, such as the finding of some defects in the PLEXIL's initial design and the development of new features for the language. Section 8.4 presents a case study about a cruise control system that is specified and formally verified in `PLX`. Some related work is discussed in Section 8.5. The source code of the rewriting logic semantics in Maude is available at

`http://camilorocha.info/thesis`.

8.1 PLEXIL Overview

This section presents an overview of PLEXIL, a synchronous language for automation developed by NASA. The reader is referred to [39] for a detailed description of the language.

A PLEXIL program, called a *plan*, is a tree of *nodes* representing a hierarchical decomposition of tasks. Interior nodes, called *list nodes*, provide control structure and naming scope for local variables. The primitive actions of a plan are specified in the leaf nodes. Leaf nodes can be *assignment nodes*, which assign values to local variables, *command nodes*, which call external commands, or *empty nodes*, which do nothing. PLEXIL plans interact with a functional layer that provides the interface with the external environment. This functional layer executes the external commands and communicates the status and result of their execution to the plan through *external variables*.

Nodes have an *execution state*, which can be *inactive*, *waiting*, *executing*, *iterationend*, *failing*, *finishing*, or *finished*, and an *execution outcome*, which can be *unknown*, *skipped*, *success*, or *failure*. They can declare local variables that are accessible to the node in which they are declared and all its descendants. In contrast to local variables, the execution state and outcome of a node are visible to all nodes in the plan. Assignment nodes also have a *priority* that is used to solve race conditions. The *internal state* of a node consists of the current values of its execution state, execution outcome, and local variables.

Each node is equipped with a set of *gate conditions* and *check conditions* that govern the execution of a plan. Gate conditions provide control flow mechanisms that react to external events. In particular, the *start condition* specifies when a node starts its execution, the *end condition* specifies when a node ends its execution, the *repeat condition* specifies when a node can repeat its execution, and the *skip condition* specifies when the execution of a node can be skipped. Check conditions are used to signal abnormal execution states of a node and they can be either *pre-condition*, *post-condition*, or *invariant* conditions. The language includes Boolean, integer and floating-point arithmetic, and string expressions. It also includes *lookup expressions* that read the value of external variables provided to the plan through the executive. Expressions appear in conditions, assignments, and arguments of

commands. Each of the basic types is extended by a special value *unknown* that can result, for example, when a lookup fails.

The execution of a plan in PLEXIL is driven by external events that trigger changes in the gate conditions. All nodes affected by a change in a gate condition synchronously respond to the event by modifying their internal state. These internal modifications may trigger more changes in gate conditions that in turn are synchronously processed until quiescence is reached for all nodes involved. External events are considered in the order in which they are received. An external event and all its cascading effects are processed before the next event is considered. This behavior is known as *run-to-completion semantics*.

Henceforth, the notation (Γ, π) is used to represent the execution state of a plan, where Γ is a set of external variables and their current values, and π is a set of nodes and their internal states. Formally, the semantics of PLEXIL is defined on states (Γ, π) by a compositional layer of five reduction relations [39]. The *atomic relation* describes the execution of an individual node in terms of state transitions triggered by changes in the environment. The *micro relation* describes the *synchronous* reduction of the atomic relation with respect to the *maximal redexes strategy*, i.e., the synchronous application of the atomic relation to the maximal set of nodes of a plan. The remaining three relations are the *quiescence relation*, the *macro relation*, and the *execution relation* that describe, respectively, the reduction of the micro relation until normalization, the interaction of a plan with the external environment upon one external event, and the n-iteration of the macro relation corresponding to n time steps.

Consider the PLEXIL plan in Figure 8.1. The plan consists of a root node `Exchange` of type *list*, and leaf nodes `SetX` and `SetY` of type *assignment*. The node `Exchange` declares two local variables `x` and `y`. The values of these variables are exchanged by the synchronous execution of the node assignments `SetX` and `SetY`. The node `Exchange` also declares a start condition and an invariant condition. The start condition states that the node can start executing whenever the value of an external variable `T` is greater than 10. The invariant condition states that at any state of execution the values of `x` and `y` add up to 3.

```
Exchange: {
    Integer x = 1;
    Integer y = 2;
    StartCondition: Lookup(T) > 10;
    Invariant: x+y == 3;
    NodeList:
        SetX: { Assignment: x = y; }
        SetY: { Assignment: y = x; }
}
```

Figure 8.1: A PLEXIL plan that reads the value of an external variable `T` and synchronously exchanges the values of internal variables `x` and `y`.

8.2 Formal Semantics

The PLEXIL rewriting logic semantics `PLX` is defined in Maude and has several modules comprising more than 2000 lines of code. A complete review of its implementation is out of the scope of this section. Instead, this section highlights some of the main features of the formal semantics and provides a high-level overview of the `PLX` source code available with the thesis at `http://camilorocha.info/thesis`. Some syntax details have been omitted or changed intentionally to favor a cleaner notation and a clearer explanation.

At the top level, the state space is represented by the top sort `Sys`, which is made up of sequences of collections of external variables and configurations of objects:

```
sort Sys .
op _|-_ : EnvList Configuration -> Sys [ctor] .
```

Sequences of collections of external variables correspond to sequences of events in the external environment; more than one event can be recorded in each collection which results in more than one external variable being updated. Similar to the IBOS formal semantics in Chapter 6, objects are made up out of an object identifier, a type an a set of attributes.

Object identifiers are nonempty lists of sort `NeQualified` made up of simple identifiers of sort `Identifier`, and obtained by instantiating Maude's parametric list sort `LIST`:

```
protecting LIST{Identifier}
  * ( op nil to nilq ,
      sort List{Identifier} to Qualified,
      sort NeList{Identifier} to NeQualified,
      op __ to _._ ) .
```

For each plan, the names of the nodes are used to populate the sort `Identifier` with constants. The elements in the list represent the hierarchy of a node in a plan. For example, the actual name of node `SetX` in Figure 8.1

is defined by two constants `SetX` and `Exchange` of sort `Identifier`, and is represented by the ground term `SetX.Exchange`. The semantics assumes that the qualified names in an input plan are *unique*.

The type of an object identifies the type of node or local memory:

```
op list       : -> Cid .
op command    : -> Cid .
op assignment : -> Cid .
op empty      : -> Cid .
op memory     : -> Cid .
op extvar     : -> Cid .
```

The execution state of nodes and local memories is represented by the attributes in its object representation. The following are some attribute names:

```
--- internal state
op status:_       : Status -> Attribute [ctor] .
--- execution outcome
op outcome:_      : Outcome -> Attribute [ctor] .
--- assignment
op _:=_           : NeQualified Expression -> Attribute [ctor] .
--- initial value of a local memory
op initVal:_      : Value -> Attribute [ctor] .
--- current value of a local memory
op actVal:_       : Value -> Attribute [ctor] .
--- priority for assignments
op priority:_     : Rat -> Attribute [ctor] .

--- Conditions
op repeatc:_      : Expression -> Attribute [ctor] .
op startc:_       : Expression -> Attribute [ctor] .
op endc:_         : Expression -> Attribute [ctor] .
op post:_         : Expression -> Attribute [ctor] .
op skip:_         : Expression -> Attribute [ctor] .
op pre:_          : Expression -> Attribute [ctor] .
op inv:_          : Expression -> Attribute [ctor] .
op exit:_         : Expression -> Attribute [ctor] .
```

The most basic expressions in `PLX` are *constant* values with sort `Value`, which are formed from Boolean, integer, float, and string Maude values. Additional constants such as `unknown` and `aborted` are introduced, respectively, to model the special information about unknown information and aborted execution.

```
sort Value .
op v : Int -> Value .
op v : Bool -> Value .
op v : Float -> Value .
op v : String -> Value .
ops unknown aborted : -> Value .
```

More general expressions are formed from constant values with constructor `const`, from local memory names with constructor `var`, and from external variable access with constructor `lookups`.

```
sort Expression .
op const   : Value -> Expression [ctor] .
op var     : NeQualified -> Expression [ctor] .
op lookup  : NeQualified -> Expression [ctor] .
```

Expressions can also be formed from the Boolean, integer, and float operators in the usual way.

As an illustration, the following is an initial state for the `Exchange` PLEXIL plan in Figure 8.1:

```
nil |-
< Exchange : list |
    status: inactive,
    outcome: none,
    startc: (lookupOnChange(T, v(0)) > const(v(10))),
    inv: (var(x . Exchange) + var(y . Exchange) equ const(v(3))), ... >
< SetX . Exchange : assignment |
    status: inactive,
    outcome: none,
    (x . Exchange) := var(y . Exchange), ... >
< SetY . Exchange : assignment |
    status: inactive,
    outcome: none,
    (y . Exchange) := var(x . Exchange), ... >
< x . Exchange : memory | initVal: v(1),actVal: v(1) >
< y . Exchange : memory | initVal: v(2),actVal: v(2) >
< T : extvar | actVal: v(15) >
```

In this state, the value of the external variable `T` is the constant integer value 15. Constant `nil` with sort environment list indicates that there are not external events besides the one for initially updating `T`.

8.2.1 Synchronous Simulation

PLEXIL's *atomic relation* is defined by 42 rules, indexed by the type and the execution status of nodes into a dozen groups. Each group associates a priority to its set of rules, which defines a linear order on the set of rules.

Atomic transitions are modeled by a collection of equations implementing the serialization procedure [91, 31]. These serialization equations are all triggered by the `micro` function:

```
op micro : Configuration -> Configuration .
```

The argument of `micro` is a configuration of objects representing the execution state of a plan. The output is a configuration representing the

synchronous application of all possible one-step atomic reductions on the input configuration under the maximal redexes strategy. Note that since PLEXIL is deterministic, it is assumed that the output of `micro` is at most one state on any given input. If a synchronous reduction is not possible, function `micro` returns the input configuration.

PLEXIL's micro relation is specified by the rewrite rule `[micro]`, which uses the function `micro`:

```
crl [micro] :
    EL:EnvList |- Cnf:Configuration
 => EL:EnvList |- Cnf':Configuration
 if Cnf':Configuration := micro(Cnf:Configuration)
 /\ Cnf:Configuration =/= Cnf':Configuration .
```

A micro reduction is possible only when the outcome of `micro` is different to its input. Function symbol '`=/=`' is Maude's built-in inequality operator. See [31] for more details on the implementation of the serialization procedure by function `micro`.

8.2.2 External Events

PLEXIL execution is driven by *external events*. The set of such events includes events related to lookup in conditions, e.g., changes in the value of an external state that affects a gate condition, acknowledgments that a command has been initialized, reception of a value returned by a command, etc.

PLEXIL's interaction with the external environment is defined by the *macro* relation. In the rewriting logic semantics `PLX`, rewrite rule `[macro]` models the interaction of a plan with the external events:

```
crl [macro] :
    Env:Environment . EL:EnvList |- Cnf:Configuration
 => EL:EnvList |- updateConf(Cnf':Configuration,Env:Environment)
 if Cnf':Configuration := micro(Cnf:Configuration)
 /\ Cnf:Configuration == Cnf':Configuration .
```

A rewrite step with rule `macro` is possible whenever there is at least an external event (modeled by `Env:Environment`) and no micro steps are possible, which corresponds to the formal definition of PLEXIL's macro relation. Auxiliary function `updateEnv` updates the value associated to the external variables in the execution state with that from the next event of the environment.

As an illustration on how rules `[micro]` and `[macro]` interact in the rewriting logic semantics `PLX`, consider the following output given by Maude's

`search` command:

```
search in EXCHANGE : compile(ExchangeEnv, Exchange) =>! X:Sys .

Solution 1 (state 9)
states: 10  rewrites: 1217 in 4ms cpu (4ms real) (304250 rewrites/second)
X:Sys -->

 nil |-
 < Exchange : list |
     status: finished,
     outcome: success,
     startc: (lookupOnChange(T, v(0)) > const(v(10))),
     inv: (var(x . Exchange) + var(y . Exchange) equ const(v(3))), ... >
 < SetX . Exchange : assignment |
     status: finished,
     outcome: success,
     (x . Exchange) := var(y . Exchange), ... >
 < SetY . Exchange : assignment |
     status: finished,
     outcome: success,
     (y . Exchange) := var(x . Exchange), ... >
 < x . Exchange : memory | initVal: v(1),actVal: v(2) >
 < y . Exchange : memory | initVal: v(2),actVal: v(1) >
 < T : extvar | actVal: v(15) >
```

In this case, plan `Exchange` is executed from an initial state, represented by `compile(ExchangeEnv, Exchange)` and that corresponds to the initial state presented before in this section (where the value of `T` is 15). The result of the search is exactly one state in which the values of `x . Exchange` and `y . Exchange` have been exchanged, the internal state of execution of the nodes is `finished`, and the outcome of their execution is `success`.

8.3 Design Validation

The rewriting logic semantics `PLX` has been used to validate the design of PLEXIL against a wide variety of plan examples. Thanks to the formal semantics two problematic issues about the original PLEXIL semantics were discovered:

Non-atomicity of the atomic relation. A prior version of the atomic rules for executing assignment nodes introduced an undesired interleaving semantics for variable assignments, which invalidated the synchronous nature of the language.

Spurious non-termination of plans. Due to lack of detail in the original specification of some predicates, there were cases in which some tran-

sitions for list nodes ending iteration would lead to spurious infinite loops.

Although the formal operational semantics of PLEXIL in [30] had been previously used to prove several properties of PLEXIL, neither one of these issues was uncovered. As as matter of fact, these issues do not compromise any of the proven properties of the language. Solutions to both issues were provided using the executable rewriting logic semantics and have been adopted in the latest version of the formal PLEXIL semantics.

The rewriting logic semantics `PLX` has also been used to study variations and extensions of the PLEXIL language design.

PLEXIL's macro relation is especially important because it is the semantic relation defining the interaction of a plan with the external environment. On the one hand, it is reasonable to have access to the external state as often as possible so that lookups in each atomic reduction can use the latest information available. On the other hand, it can be computationally expensive to implement such a policy because sensors or similar artifacts can significantly delay the execution of a plan. Another dimension of the problem arises when a guard of an internal loop depends on external variables: should the loop run-to-completion regardless of the possible updates to the value of the variable in its guard, or should it stop at each iteration so that the value of the external variable can be updated?

The rewriting logic semantics `PLX` has been modified to accommodate alternative specifications of PLEXIL's semantics with different definitions of the macro relation. These semantic variants of PLEXIL have been studied and have been exercised with a significant number of examples.

Another concrete example that illustrates the use of PLEXIL's rewriting logic semantics by the designers of the language is the addition of a gate condition called *exit condition*. The exit condition provides a mechanism for a clean interruption of execution. In order to support this feature, the `PLX` specification of the atomic relation was modified to include the intended semantics. Given the modular definition of the formal semantics none of the other rewriting relations need to be modified.

8.4 A Case Study

A cruise control system adapted from [13] is presented to showcase the model checking capabilities made possible by the `PLX` executable semantics. Originally, the cruise control model was designed for the Enhanced Oper-

ator Function Model (EOFM) formalism, which is intended for the study of human behavior in a human-computer interaction framework. However, PLEXIL shares many characteristics with EOFM, including the hierarchical structure of tasks decomposed into sub-tasks and the execution governed by conditions (*pre*, *post*, *repeat*, *invariant*).

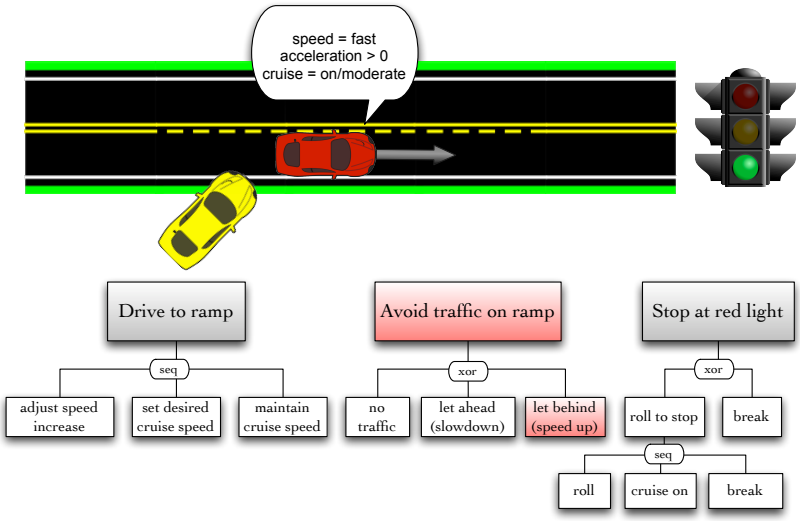

Figure 8.2: Cruise control model with task hierarchy.

8.4.1 Model Description

The cruise control model consists of three main components: car, driver, and stoplight, which execute synchronously. The operator drives the car on a street, approaching the stoplight. Other cars may merge into the lane from a side ramp, roughly midway through. The car has three controls represented in the model: the gas and break pedals to manage speed and acceleration, and a cruise button to switch the cruise mode on/off and set the cruise speed. The human operator's plan is to safely operate the controls of the car to achieve three sub-goals: (i) drive at a desired cruise speed, (ii) avoid the possible merging traffic from the ramp, and (iii) obey the traffic light at the intersection, i.e., stop the car in time if the light turns red. All three properties can be represented in PLEXIL. Here the focus is on the third, which is a safety property.

The model parameters are: the geometry of the intersection, i.e., the length of each street segment; the location of the ramp along the street, in distance units; the stoplight cycle length, in time units, for each color; and the speed range, in distance per time units.

Model variables. The model variables and their range are selected according to an abstraction scheme that discretizes the values to allow finite state model checking, yet leaves sufficient information to make the study relevant.

- $\texttt{distance} \in [0 \ldots 55]$, the distance of the car to the intersection;
- $\texttt{time} \in [0 \ldots 28]$;
- $\texttt{speed} \in \{\texttt{stopped} = 0, \texttt{slow} = 1, \texttt{moderate} = 2, \texttt{fast} = 3\}$;
- $\texttt{acceleration} \in \{-1, 0, 1\}$;
- $\texttt{cruise\_enabled} \in \{\mathit{true}, \mathit{false}\}$;
- $\texttt{cruise\_speed} \in \{0, 1, 2, 3\}$;

Transitions. The car advances according to its speed until it reaches the intersection. It updates

$$\texttt{distance} := \texttt{distance} - \texttt{speed} * \texttt{timestep}$$

while the condition $\texttt{speed} > 0 \wedge \texttt{distance} > 0$ holds. The discretized speed can change by at most one unit at a time, hence the possible values for acceleration are only $\{-1, 0, 1\}$. The stoplight counts down the time units until the end of the green-yellow-red cycle by assigning

$$\texttt{stoplight} := \texttt{stoplight} - \texttt{timestep}.$$

The light is red in the time interval $[0 \ldots 8]$, yellow in $[9 \ldots 12]$, and green in $[13 \ldots 28]$.

The complexity resides in capturing the decision making of the driver. In the first segment, the driver wants to set the cruise control to a desired speed (e.g., `moderate`). The driver has the choice of accelerating from `slow` or decelerating from `fast`, and then enable the cruise control which will maintain the desired speed. On the second segment, the driver needs to react to merging traffic from the ramp. If any car is on the ramp, the driver may choose to let the other car go in front by slowing down, or to leave it behind by speeding up. On the last segment, the driver has to react to the stoplight turning red. The driver may choose to maintain the speed and then break before reaching the stoplight, or may roll to a stop by releasing the gas pedal.

Comparison with the EOFM model.

- The original abstraction has been refined in PLEXIL to allow more distance and time divisions, making it more realistic; in the EOFM model the distance is heavily discretized (abstract locations 0 to 7) and not coordinated with the time to travel each segment.
- Non-determinism is introduced by lookups of environment variables. The script plays out a sequence of random choices for three Boolean environment variables: `MergingTraffic`, `LetBehind`, `RollStop`.
- Some of the concepts are essentially cognitive in nature, as they depend on the subjective (sometimes erroneous) perceptions and assessments of the situation by the human operator, hence they cannot be as naturally captured in the formal model. However, both normative and erroneous behaviors are captured in the PLEXIL model, and it is the job of the model checker to discover violations.
- The synchronous behavior is natural in PLEXIL, no further instrumentation is necessary, while in EOFM synchrony has to be expressly specified, using appropriate decomposition operators.

8.4.2 Verification

The property of interest can be expressed either as a global invariant in the PLEXIL model itself and checked with the generic "check invariants" button, or entered in the LTL Model Checking dialog window. The safety property is specified in the top level task node `Main` as the invariant condition:

$$\neg(\texttt{stoplight} \leq \texttt{red} \wedge \texttt{distance} = 0 \wedge \texttt{speed} > 0),$$

stating that it is not the case that the vehicle is moving at the intersection when the light is red.

The PLEXIL semantics shows that the execution of the plan ends with the outcome `invariantFail` for the root node (and `parentFail` for the successor nodes) when the environment variables `MergingTraffic`, `LetBehind`, and `RollStop` are all true. The result of model checking the safety property is an execution trace where the formula is violated.

The counterexample can be described as follows:

1. the car enters at `low` speed at $\texttt{distance} = 55$ and $\texttt{time} = 28$;

2. the driver accelerates to the desired `moderate` speed and sets the cruise on at `time` = 20 and `distance` = 42;

3. at the ramp, with `distance` = 33, the driver decides to let the merging car behind by accelerating to `fast` at `time` = 12 and `distance` = 25;

4. the stoplight light turns yellow, the driver chooses to roll to a stop (assessing there is sufficient distance to the intersection to do so, by releasing the gas pedal);

5. with the acceleration negative, the driver does not disengage the cruise mode, the cruise control kicks in and maintains the cruise `speed` to `moderate` for one execution cycle at `time` = 6 and `distance` = 10;

6. the effect of the automation is that the (now necessary) breaking is too late to decrease the speed from `moderate` to `low` at `time` = 2 and `distance` = 2, and then `stopped` in two execution cycles; and

7. when time expires, the car is moving in the intersection on the red light.

To correct the problem, the node corresponding to the "roll to stop" action has to be rectified, in order to include a check on the status of the cruise control. The driver either has to make sure it is disabled before initiating the "roll to stop" option or should manually disable it. In PLEXIL, this can be instrumented via a *start condition* or, by duality, with the corresponding negated *skip condition*. No other combination of environment lookup variables leads to violations in this model.

The full model of the cruise control system consists of 929 lines of code.

8.5 Related Work and Concluding Remarks

Rewriting logic has been used previously as a testbed for specifying and animating the semantics of synchronous languages. M. AlTurki and J. Meseguer [1] have studied the rewriting logic semantics of the language Orc, which includes a synchronous reduction relation. T. Serbanuta *et al.* [97] define the execution of P-systems with structured data with continuations. The focus of the former is to use rewriting logic to study the (mainly) nondeterministic behavior of Orc programs, while the focus of the latter is to study the relationship between P-systems and the existing continuation

framework for enriching each with the strong features of the other. The approach here is instead based on exploiting the determinism of a synchronous relation to tackle the problem associated with the interleaving semantics of concurrency in rewriting logic. P. Lucanu [66] studies the problem of the interleaving semantics of concurrency in rewriting logic for synchronous systems from the perspective of P-systems. The determinism property of the synchronous language Esterel [11] was formally proven by O. Tardieu in [102]. C. Rocha and C. Muñoz [92] have implemented a framework for the simulation of synchronous systems in Maude, following the main ideas presented in this chapter for the rewriting logic semantics of PLEXIL.

An executable semantics of PLEXIL has been developed by P. J. Strauss in the Haskell language [99] with the aim of analyzing features of the language regarding the plan interaction with the environment. As a result, new data types representing the external world have been proposed for more dynamic runtime behavior of PLEXIL plans. More recently, D. Balasubramanian et al. have proposed Polyglot, a framework for modeling and analyzing multiple Statechart formalisms, and have initiated research towards the formal analysis of a Statechart-based semantics of PLEXIL [7].

CHAPTER 9

SYMBOLIC REACHABILITY FOR PLEXIL MODULO INTEGER CONSTRAINTS

In Chapter 8 it was explained that the execution of a plan in PLEXIL is driven by external events that trigger changes in the gate conditions. This chapter presents a case study on the symbolic analysis of reachability properties for PLEXIL with the main goal of complementing the formal verification capabilities already available to PLEXIL with the rewriting logic semantics `PLX` presented in Chapter 8. As a result, this chapter reports on the implementation of a *symbolic* rewriting logic semantics for a large subset of the PLEXIL language that is able to automatically detect reachability violations on input plans where the *values of external variables can be left unspecified*.

The notion of an invariant violation or a race condition is important in PLEXIL. In particular, as a safety property of the language, a plan is considered *invalid* if there is an execution that leads to a race condition. In other words, a plan can be considered *safe* if none of its possible executions leads to a race condition. Of course, detecting a race condition when knowing in advance what the values of the external variables is not very difficult. But what is required is to ensure that the *non-determinism* introduced by the external environment can never lead to a race condition.

In order to motivate the discussion, consider the plan `AssignWithConflig` in Figure 9.1. This plan has one list node and two assignment nodes, `NonNeg` and `NonPos`. It declares a local integer memory `x` and interacts with the external environment via the integer variable `S`. Note that depending on the value of `S`, the assignment nodes `NonNeg` and `NonPos` may or may not start execution, and a race condition can happen on `x` when the value of `S` is 0. With the symbolic semantics presented in this chapter, the race condition on

```
AssignWithConflict: {
  Integer       x = 0;
  Invariant:    x >= 0;
  NodeList:
  NonNeg: {
    Start:      Lookup(S) >= 0;
    Assignment: x := 1;
  }
  NonPos: {
    Start:      Lookup(S) <= 0;
    Assignment: x := 2;
  }
}
```

Figure 9.1: A parallel assignment with a potential race condition.

`x` can be automatically detected. In contrast, such an automatic checking is not possible with the rewriting logic semantics `PLX` because of the inherently symbolic interpretation of `S`.

This chapter uses the techniques developed in Chapter 7 to study the formal verification of symbolic reachability properties for PLEXIL. The verification task focuses on analyzing the non-determinism introduced in the language by lookups of the external environment with help of Maude's `search` command and the LTL model checker. The symbolic semantics is given in the form of a constrained rewrite theory that specifies the symbolic atomic behavior of the language, which is encoded and executed in the Maude system extended with CVC3 (available from the Matching Logic Project [93]). The external variables of the environment are modeled as Boolean and integer constrained built-in terms and CVC3's decision procedure for quantifier-free linear integer arithmetic is used as an oracle for solving constraints on them.

Given the asynchronous nature of the rewriting relations directly executable in Maude, the synchronous execution of the symbolic atomic relation is obtained by exploiting rewriting logic's reflective capabilities available in Maude. It is important to mention that the symbolic semantics of PLEXIL uses data structures based on lists, sets, and multisets. Hence the insistence in Chapter 7 on supporting any combination of associativity, commutativity, and identity axioms in the specifications.

This chapter is organized as follows. Section 9.1 explains how symbolic states are modeled in the symbolic semantics. Section 9.2 presents an summary on how the symbolic atomic relation of PLEXIL is encoded as

a rewrite theory in Maude and Section 9.3 explains how its synchronous execution is obtained. Section 9.4 shows how the symbolic semantics can be used to detect the race condition in the running example, and also find an automatic proof of its absence in other cases. The source code of the symbolic rewriting logic semantics in Maude is available at `http://camilorocha.info/thesis`.

9.1 Symbolic States

At the top level, a state is represented by the top sort `Sys`, whose terms are made up of a Boolean constraint and a configuration of objects:

```
sort Sys .
op {_,_} : iBool Configuration -> Sys [ctor frozen(2)] .
```

With respect to the development in Section 7.1, sort `Sys` instantiates the abstract top sort `s` of constrained terms. Objects in the symbolic semantics obey the syntax of objects already introduced in chapters 6 and 8.

Sort `iBool` represents built-in Boolean terms that are used as Boolean constraints. Similarly, sort `iInt` represents built-in integer terms that can be part of the configuration. Note that in contrast to the ground semantics of PLEXIL presented in Section 8.2, the external environment is assumed to be completely encoded in the configuration of objects in the symbolic semantics. Also note that the second argument in a constrained state has a frozenness constraint (see Chapter 3 for details). This is to prevent any direct rewrite with the rewrite rules defining PLEXIL's symbolic atomic relation. As it will be explained, these rules are not executed directly but by means of Maude's reflective capabilities.

Terms of sort `iBool` and `iInt` represent symbolic built-in expressions. They include the following definitions:

```
sort iBool iInt .

op c : Bool -> iBool [ctor] .
op c : Int  -> iInt [ctor] .

op b : Nat -> iBool [ctor] .
op i : Nat -> iInt [ctor] .
```

Constructor function symbol `c` is a wrapper for Boolean and integer values. Constructor function symbols `b` and `i` play a key role in the symbolic semantics: their goal is to encode, respectively, PLEXIL's Boolean and integer variables. In this way, a Boolean variable x_1 in PLEXIL could be

expressed as the ground term `b(1)` of sort `iBool`. The overall picture is the following: the set X_Λ in Chapter 7, for the case of PLEXIL's Boolean and integer variables, is actually being encoded in the symbolic semantics by ground terms of sort `iBool` and `iInt`, respectively.

The Boolean and integer sorts `iBool` and `iInt` include more operators. For instance, they include memory binding operators with the function symbols `bmem` and `imem`, lookup expressions with the function symbols `blookup` and `ilookup`, and the usual operators with self-explanatory syntax:

```
op bmem    : NeQualName -> iBool .
op blookup : NeQualName -> iBool .

op imem    : NeQualName -> iInt .
op ilookup : NeQualName -> iInt .

op -_ : iInt -> iInt .
ops _+_ _*_ : iInt iInt -> iInt [assoc comm] .
op _-_ : iInt iInt -> iInt .

ops _<=_ _<_ _>=_ _>_ : iInt iInt -> iBool .
ops _===_ _=//=_ : iInt iInt -> iBool [comm] .
```

It is important to note that the operators '`===`' and '`=//=`' shown above are actually equality enrichments for the sort `iInt`. There are similar operators for the sort `iBool`.

Boolean and integer expressions can be evaluated 'symbolically' by means of function `eval`:

```
op eval : Configuration iBool -> iBool .
op eval : Configuration iInt -> iInt .
```

The evaluation of an expression by `eval` is given w.r.t. an object configuration and it is equationally defined recursively on the complexity of expressions.

For instance, the following is an equation evaluating a Boolean lookup of an external variable:

```
eq eval( (< NeQN:NeQualified : extvar |
             type: boolean,
             values: (iB:iBool ; iEL:iBoolList),
             AtS:AttributeSet > Cnf:Configuration),
         blookup(NeQN:NeQualified))
 = iB:iBool .
```

Attribute name `type` is used to identify the type (either `boolean` or `int`) of an external variable. Attribute name `values` is used as a placeholder for lists of expressions (either of sort `iBoolList` or `iIntList`) that represent the future values of lookups.

9.2 The Symbolic Atomic Relation

The symbolic atomic relation of PLEXIL is specified by a set of constrained rewrite rules encoded as rewrite rules in an ordinary rewrite theory. The notion of guarded actions is useful for this purpose, as explained below.

A symbolic atomic transition can update the status and the outcome of a node, update the value of a memory, and reset the value of a memory to its initial value. These actions are modeled by sort `Action` as follows:

```
sort Action .
op set-status  : NeQualName Status -> Action [ctor] .
op set-outcome : NeQualName Outcome -> Action [ctor] .
op set-value   : NeQualName iBool -> Action [ctor] .
op set-value   : NeQualName iInt -> Action [ctor] .
op reset       : NeQualName -> Action [ctor] .
```

Sets of actions are modeled by sort `ActionSet` that is defined by instantiating Maude's parametric module `SET`:

```
pr SET{Action} * (sort NeSet{Action} to NeActionSet,
                  sort Set{Action} to ActionSet,
                  op empty to mtas,
                  op _,_ to _;_) .
```

A guarded set of actions represents actions constrained by a Boolean constraint. They are a convenient representation for the purpose of encoding the constrained rules:

```
pr 4TUPLE{String,Nat,iBool,ActionSet}
   * (sort Tuple{String,Nat,iBool,ActionSet} to 4GuardedActionSet) .
```

In order to motivate the usefulness of guarded actions, consider the following rewrite rules encoding PLEXIL's atomic transitions for lists in state *executing* corresponding to the transition diagram depicted in Figure 9.2:

```
rl [exec-list-1] :
   < O:NeQualified : list | status: executing,
        AtS:AttributeSet >
=> ("exec-list",
    1,
    anc-inv(O:NeQualified) === c(false),
    ( set-outcome(O:NeQualified,fail(parent)) ;
      set-status(O:NeQualified,failing))) .

rl [exec-list-2] :
   < O:NeQualified : list | status: executing,
        inv: iB:iBool,
        AtS:AttributeSet >
=> ("exec-list",
    2,
```

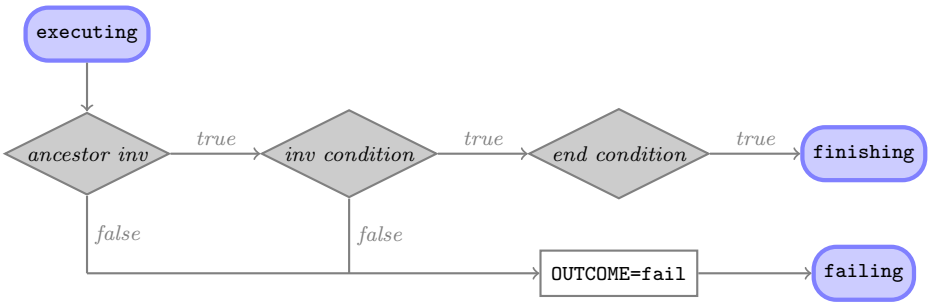

Figure 9.2: Atomic transitions for list nodes in state *executing*.

```
    iB === c(false),
    ( set-outcome(O:NeQualified,fail(inv)) ;
      set-status(O:NeQualified,failing))) .

rl [exec-list-3] :
   < O:NeQualified : list | status: executing,
        end: iB:iBool,
        AtS:AttributeSet >
=> ("exec-list",
    3,
    iB:iBool,
    set-status(O:NeQualified,finishing)) .
```

First note that all terms representing guarded actions occur in the right-hand side of the rewrite rules. Second, the string argument is the same in all four tuples and the second argument records the order in which the conditions appear is the diagram. This two values are used by the instrumentation of the serialization procedure, presented in the next section, to compute the set of maximal redexes. Actions with higher priority (indicated here by a smaller value) have precedence over actions with lower priority (indicated here by a larger value). The third argument is the actual guard of the set of actions that appear in the fourth argument.

In order to understand how the symbolic atomic semantics works based on this encoding, consider a ground state {`B,Cnf`} of sort `Sys` having the following objects as part of its object configuration:

```
< bar : list |
    inv: b(0), ... >

< foo . bar : list |
    status: executing,
    inv: b(0) == false,
    end: b(0), ... >
```

Nodes `bar` and `foo.bar` are list nodes; the former is the parent of the

latter. The invariant of node `bar` holds whenever the value (of the Boolean variable represented) by `b(0)` is true. The invariant of `foo.bar` holds whenever `b(0)` is false. The end condition of `foo.bar` holds whenever `b(0)` is true. Then, no matter what Boolean condition the constraint `B` represents, node `foo.bar` can never transition with rule `[exec-list-3]` because `[exec-list-1]` has higher priority. However, there may be cases when both rules `[exec-list-1]` and `[exec-list-2]` can be applied but not in conjunction because they are mutually exclusive.

9.3 Synchronous Symbolic Execution

The problem of simulating PLEXIL's symbolic synchronous relation can be decomposed into a combination of smaller problems. The approach has three phases: matching, parallel maximization, and action application.

In the *matching* phase, all *possible* guarded actions asynchronously induced by the encoding of the symbolic atomic rules on a symbolic state are collected. This phase is performed using a mapping:

```
op match : Sys -> PreSysSet .
```

The sort `PreSysSet` is introduced to represent multisets of constrained actions. On input {`B,Cnf`}, every constrained set of actions in `match({B,Cnf})` is such that its constraint is satisfiable in conjunction with the constraint `B`. The implementation of *match* uses Maude metalevel facilities to compute the guarded actions by applying meta-rewrites on `Cnf` with each atomic rule.

The *parallel maximization* phase computes a collection of constrained actions from the collection of constrained actions returned by `match`. Each of the constrained actions returned in this phase represents a maximal parallel reduction of atomic transitions whose constraints are all satisfiable in conjunction with the input constraint. This phase is performed using a mapping:

```
op maxp : iBool PreSysSet -> PreSysSet .
```

In the *action application* phase, constrained actions are applied to the input constrained term, representing a symbolic maximal parallel atomic reduction. This is done for each constrained action obtained from the parallel maximization phase on the initial input, thus capturing the (possible) non-deterministic behavior of the symbolic parallel reduction. This phase is performed using a mapping:

```
op apply : Sys PreSys -> Sys .
```

The call to functions `match`, `maxp`, and `apply`, is triggered by function `bgc`:

```
op bgc : Sys -> SysSet .
```

Given a ground term {`B,Cnf`} of sort `Sys`, function `bgc` computes a set of ground terms of sort `Sys`, namely, those states reachable from {`B,Cnf`} in one symbolic micro step.

The following are the rewrite rules that specify the symbolic micro and macro relations in the symbolic semantics of PLEXIL:

```
crl [micro] :
    St => { iB, Cnf }
 if (iB,Cnf),GCS := bgc(St) .

crl [macro] :
    St => { coll-constr(pop-env(conf(St))), pop-env(conf(St)) }
 if mt := bgc(St)
 /\ has-future?(conf(St)) = true .
```

Term `mt` is a constant with sort `SysSet` denoting the empty set of states. Function `has-future?` checks if there is an event waiting to be processed. Functions `coll-constr` and `pop-env` are auxiliary functions that update the values of the external variables in the object configuration.

9.4 Symbolic LTL Model Checking

Detection of race conditions on local memories and violation of node invariants are important in PLEXIL. As such, predicates for checking them are already available from the symbolic semantics. In particular, states predicates `inv` and `race-free`, which take an argument of sort `NeQualified`, are offered to the user.

The intended semantics of the state predicates is with respect to the initial semantics of PLEXIL. For example, consider the following definition of `inv` in the syntax of Maude model checker:

```
eq ({ iB:Bool,
      < O:NeQualified : C:Cid | inv: iB':iBool, AtS:AttributeSet >
      Cnf:Configuration }) |= inv(O:NeQualified)
 = check-unsat(iB:iBool and
               not(eval(< O:NeQualified : C:Cid |
                           inv: iB':iBool, AtS:AttributeSet > Cnf,
                        iB':iBool))) .
```

The invariant condition of node `O` represented by the Boolean expression `iB'` yields an invariant violation for `O` whenever the conjunction of the state's

constraint `iB` and the negation of `iB'` is unsatisfiable. This precisely means that there is a ground counter-example state for the invariance of the node. Function `check-unsat` implements the call to CVC3:

```
op check-unsat : iBool -> Bool .
```

Recall the plan `AssignWithConflict` in Figure 9.1, which has a potential race condition for the local memory `x`. Assume that `SPLX` represents the symbolic rewriting logic semantics of PLEXIL, and let `init` be a configuration of objects representing an initial configuration for `AssignWithConflict`. Consider the following safety verification requirements:

$$\mathcal{T}_{\texttt{SPLX}}, \{\texttt{c(true)}, \texttt{init}\} \models \Box\texttt{race-free(x.AssignWithConflict)}, \tag{9.1}$$

$$\mathcal{T}_{\texttt{SPLX}}, \{\texttt{i(0) >= c(1)}, \texttt{init}\} \models \Box\texttt{race-free(x.AssignWithConflict)}, \tag{9.2}$$

$$\mathcal{T}_{\texttt{SPLX}}, \{\texttt{i(0) >= c(1)}, \texttt{init}\} \models \Box\texttt{inv(AssignWithConflict)}. \tag{9.3}$$

The external variable `S` in `AssignWithConflict` is represented by the Boolean term `i(0)`. Property (9.1) states that there is no race condition on memory `x` if `i(0)` has no initial constraints. Property (9.2) states that there is no race condition on memory `x` if `i(0)` is assumed to be at least 1. Property (9.3) states that the invariant condition of node `AssignWithConflict` holds if `i(0)` is assumed to be at least 1. Note that these properties are symbolic reachability requirements because of the nature of the external variable `S`. Also, the constrained terms defining the initial states in these properties represent, in each case, infinitely many initial states.

By directly using Maude's LTL Model Checker, property (9.1) can be disproved, and properties (9.2) and (9.3) can be proved automatically.

```
==========================================
reduce in ASSIGNWITHCONFLICT :
  verify-lite({c(true), init}, [] race-free(x . AssignWithConflict)) .
rewrites: 2590 in 525ms cpu (1629ms real) (4929 rewrites/second)
result Bool: false
==========================================
reduce in ASSIGNWITHCONFLICT :
  verify-lite( { i(0) >= c(1), init}, [] race-free(x . AssignWithConflict)) .
rewrites: 2846 in 575ms cpu (614ms real) (4947 rewrites/second)
result Bool: true
==========================================
reduce in ASSIGNWITHCONFLICT :
  verify-lite( {i(0) >= c(1), init}, [] inv(AssignWithConflict) .
rewrites: 3191 in 576ms cpu (702ms real) (5534 rewrites/second)
result Bool: true
```

Function `verify-lite` is a wrapper to Maude's LTL Model Checker func-

tion `modelCheck`. This mapping outputs either `true` or `false` depending on the output of the model checker function, omitting a counterexample if any.

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

This chapter presents the conclusions of this dissertation, followed by a discussion on research directions opened by it.

10.1 Conclusions

The focus of this dissertation has been on developing *three* deductive techniques, with corresponding tool support, for symbolically reasoning about the model theoretic satisfaction relation

$$\mathcal{T}_{\mathcal{R}} \models \varphi,$$

where $\mathcal{T}_{\mathcal{R}}$ is the initial reachability model associated to an order-sorted rewrite theory $\mathcal{R}$, and φ is a reachability property. These deductive techniques for reasoning symbolically about specifications with initial model semantics include: (i) new constructor-based notions for inductive reasoning about reachability properties, (ii) a proof system for the task of proving safety properties, and (iii) a novel method for symbolic reachability analysis of rewrite theories with constrained built-ins.

The development on constructor-based notions for reachability analysis combines and generalizes two different research strands. On the one hand, it can be seen as a natural generalization from the case of equations E to that of both equations E and rules R, of the work in [54, 52] on (propositional) equational tree automata methods for checking sufficient completeness of left-linear equations modulo axioms for context-sensitive order-sorted specifications. On the other hand, it also generalizes the work by I. Gnaedig and H. Kirchner [44] on constructors for non-terminating rewrite systems in

the following precise sense: the notion of sufficient completeness proposed in [44] exactly corresponds to that of $\mathcal{R}$-sufficient completeness in this work for the special case of a rewrite theory $\mathcal{R} = (\Sigma, \varnothing, R)$, where Σ has a single sort and there are no equations. The treatment of the more general case of rewrite theories $\mathcal{R} = (\Sigma, E \cup B, R, \nu)$ clarifies the important distinction between constructors for equations and constructors for rules, extends the ideas to the more general order-sorted case modulo axioms and with frozenness information, provides new tree automata *automated* techniques that complement the deductive narrowing-based techniques proposed in [44], and, to the best of the author's knowledge, investigates for the first time the use of $\mathcal{R}$-constructors (and E-constructors) for inductive proofs of ground reachability. All these new sufficient completeness methods have been automated in a new version of the Maude Sufficient Completeness Checker (SCC).

The generic approach for the task of proving inductive safety properties, namely ground stability and ground invariance, is both *transformational* and *reductionistic*. The approach is transformational in the sense that the rules of inference transform pairs of the form $\mathcal{R} \Vdash \varphi$ into other such pairs $\mathcal{R}' \Vdash \varphi'$. It is also *reductionistic* in the sense that: (i) all safety formulas in temporal logic eventually disappear and are replaced by *purely equational formulas*, and (ii) the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is eventually replaced by its underlying *equational theory* (Σ, E). That is, in the end *all formal reasoning about safety properties is reduced to inductive reasoning about equational properties* in the underlying equational theory (Σ, E). This allows these generic safety verification methods to take advantage of the existing wealth of equational reasoning techniques and tools already available. The Maude Invariant Analyzer (InvA) tool supporting the inference system, makes systematic use of *narrowing modulo axioms* with the equations defining state predicates, to greatly simplify the equational proof obligations to which all proofs of safety formulas are ultimately reduced. It also takes full advantage of other heuristics for discharging automatically many proof obligations. The advantage of generic verification methods and tools is that the costly tool development effort can be amortized across a much wider range of applications, whereas a language-specific verification tool can only be applied to systems programmed in that specific language; two case studies on the mechanical verification of safety properties for a communication protocol and for a web browsing system are visible evidence of this fact.

The method for symbolic reachability analysis of rewrite theories with constrained built-ins is an extension of rewriting logic theories in which the rewrite rules can be used to *rewrite* terms with *constrained built-ins*. That

is, terms involving user-definable data structures, but whose only variables range over decidable domains, and constraints over these variables. The main advantage of this approach is that, under some mild syntactic conditions and the availability of an oracle for the constraints such as an SMT solver, rewrite theories with constrained built-ins can symbolically be executed by using matching instead of unification, and thus induce a rewrite relation that is amenable, for example, to model checking verification using Maude's `search` command and LTL model checker. This method can be used to address symbolic reachability problems for a broad class of specifications: its applicability is only restricted in practice by the availability of decision procedures available from the SMT solver of choice. Also, this method can be applied to solve symbolic reachability problems where the narrowing-based approach may not be applicable (e.g., associativity axioms may prove problematic because of the possible non-finiteness of a complete set of unifiers). As a case study, a rewrite theory with constrained built-ins is used to address the formal verification of some symbolic reachability properties for the Plan Execution Interchange Language (PLEXIL), a NASA language for robotic machines. The verification task focuses on analyzing the non-determinism introduced in the language by lookups of the external environment with help of Maude's `search` command and the LTL model checker. The symbolic semantics is executed in the Maude system extended with CVC3 (available from the Matching Logic Project [93]). The external variables of the environment are modeled as Boolean and integer constrained built-in terms and CVC3's decision procedure for quantifier-free linear integer arithmetic is used as an oracle for solving constraints on them.

Together, the three above-mentioned techniques, their implementation in the Maude system, and the case studies comprise a significant step forward in automatic and semi-automatic reasoning for reachability properties of rewriting logic specifications, a major research goal in the current frontier of rewriting logic research (see, e.g., [72, p. 49]).

10.2 Future Work

Rewrite theories are formal specifications of concurrent systems [70]. This dissertation suggests new open problems in the development of new deductive techniques, with tool support, for symbolically reasoning about reachability properties of rewrite theories with initial semantics. Specifically, it suggests the advancement of the following research directions.

First, since the main goal of the constructor-based notions for reachability analysis was to obtain *automatic* techniques for checking the sufficient completeness of a rewrite theory, some restrictions have been imposed, such as treating only the order-sorted case (leaving out the case of membership equational theories), and also assuming that equations and rules are left-linear and unconditional. The notion of a sufficiently complete rewrite theory is equally meaningful and useful without these restrictions. Therefore, reasoning techniques that will allow establishing such a property for more general rewrite theories should be investigated, even if such techniques are no longer automatic. The related topic of constructor-based inductive techniques for ground reachability and ground joinability has only been sketched out; it deserves a much fuller development in future work, in which a detailed comparison with alternative approaches to proving such properties should also be given. Furthermore, these constructor-based induction techniques should be supported by tools such as, for example, an extension of the current Maude Inductive Theorem Prover (ITP) and adopted to enhance the effectiveness of Maude's Church-Rosser Checker (CRC) tool.

Second, the generic approach for the task of proving inductive safety properties can be complemented by bounded symbolic execution, achieved by narrowing modulo, so that a property can be symbolically tested before trying to prove it invariant [94, 38]. In general, it is worth pursuing extensions of narrowing-based symbolic model checking techniques for conditional rewrite theories, so that these approaches can be combined for symbolic model checking and for symbolic simulation for the task of proving/disproving inductive safety properties. The mechanical verification of invariants can be more effective in InvA by adding automatic support for the discovery of invariants, improving proof heuristics and adding subsumption checks, enhancing the management of proof obligations, and extending the techniques available to the user in tools such as cover-set induction modulo axioms in the ITP.

Third, the symbolic reachability analysis with constrained built-ins should be complemented with narrowing-based techniques enhanced with SMT solving capabilities such as those recently developed by S. Escobar, V. Ganesh, and J. Meseguer. Foundations and efficient implementations are needed to handle more general and larger case studies. The symbolic folding techniques in [38] should be investigated for the case of rewrite theories with constrained built-ins. One major technical challenge is to develop a rich interaction interface from Maude to different SMT solvers. This interface should include support for obtaining models and proofs at the level of

Maude, in addition to satisfiability/unsatisfiability queries. Another interesting research question is that of how to deal generically and automatically with order-sorted built-in specifications when using decision procedures that are only many sorted. This will be useful for many Maude specifications where, for example, the use of natural numbers may in some cases be preferred and may facilitate easier expression of some problems than the use of the entire integer domain, or both naturals and integers may be used for different subproblems.

The advancement of the above-mentioned research directions seems to be an exciting topic for further research.

APPENDIX A

MISSING PROOFS FOR CHAPTER 3

This appendix presents proofs that have been omitted in Chapter 3. It includes the existence and correctness proofs of the PTA in theorems 3 and 4, and the mechanical proofs supporting the claims about the `BAG-CHOICE+CARD` and `CHANNEL` specifications.

A.1 PTA Proofs

Theorem 3. Let $\mathcal{R} = (\Sigma, E \cup A, R, \nu)$ be an admissible, ground weakly-terminating, and ground sort-decreasing simple generalized rewrite theory, and let (Υ, Ω) be a constructor signature pair for $\mathcal{R}$. If $\mathcal{R}_E$ and $\mathcal{R}_{E \cup R}$ are PTA-checkable, then there are PTAs $\mathcal{B}_E$ and $\mathcal{B}_{E \cup R}$ such that $\mathcal{R}$ is canonically sufficiently complete relative to (Ω, Υ) if and only if $L(\mathcal{B}_E) \cup L(\mathcal{B}_{E \cup R}) = \varnothing$.

Proof. First, the construction of $\mathcal{B}_{E \cup R}$ is shown in detail. A *s-context* is a term C in which a subterm t of sort s has been replaced by a *s-hole*, here signified by $\Box_s$. The symbol $\Box$ is the simplest context. If C is a s-context and t is a term of sort s, then $C[t]$ indicates C with $\Box_s$ replaced by t. At the same time, $C[t]$ indicates that the term C contains an occurrence of the subterm t. Let $\Sigma = (S, \leq, F)$, and define $\Sigma^K = (K, F^K)$ and $I_{E \cup R} = \{t \mid C[t] \in \mathit{lhs}(\overrightarrow{E} \cup R) \wedge t \notin X \wedge C \neq \Box\}$, where $\mathit{lhs}(R')$ denotes the left-hand sides of the rules R'. Construct $\mathcal{B}_{E \cup R} = (\Sigma^K, Q, \Gamma_{\mathrm{CSC}}, \Delta_{E \cup R})$ as follows:

- $Q = \{Q_k\}_{k \in K}$ with $Q_k = \{r_k\} \cup \{c_s \mid s \in S\} \cup \{p_s \mid s \in S\} \cup \{p_u \mid u \in I_{E \cup R} \cap T_{\Sigma,k}\}$,
- $\Gamma_{\mathrm{CSC}} = \{\gamma_k\}_{k \in K}$ with $\gamma_k = \neg r_k \wedge \bigvee_{s \in [k]_\leq} p_s \wedge \neg c_s$, and

- $\Delta_{E\cup R} = \{f(p_{s_1},\ldots,p_{s_n}) \to p_s \mid f \in F_{s_1\ldots s_n,s}\}$
  $\cup\ \{c(\mathrm{rep}_c^1(s_1),\ldots,\mathrm{rep}_c^n(s_n)) \to c_s \mid c \in \Upsilon_{s_1\ldots s_n,s}\}$
  $\cup\ \{p_s \to p_{s'} \mid s,s' \in S \wedge s < s'\}$
  $\cup\ \{c_s \to c_{s'} \mid s,s' \in S \wedge s < s'\}$
  $\cup\ \{f(p_{t_1},\ldots,p_{t_n}) \to p_{f(t_1,\ldots,t_n)} \mid f(t_1,\ldots,t_n) \in I_{E\cup R}\}$
  $\cup\ \{f(p_{t_1},\ldots,p_{t_n}) \to r_k \mid f(t_1,\ldots,t_n) \in T_{\Sigma,k} \cap \mathit{lhs}(\overrightarrow{E} \cup R)\}$
  $\cup\ \{f(p_{k_1},\ldots,r_{k_i},\ldots,p_{k_n}) \to r_k \mid f \in F^K_{k_1\ldots k_n,k} \wedge i \in \overline{\nu}(f)\}$,

  where $\mathrm{rep}_c^i(s)$ equals c_s if $i \in \overline{\nu}(c)$, and p_s otherwise.

Let $t \in T_\Sigma$. By structural induction on t, it follows that $t \to_{\mathcal{B}_{E\cup R}} p_s$ if and only if $t \in T_{\Sigma,s}$. A similar inductive argument shows that $t \to_{\mathcal{B}_{E\cup R}} p_u$ if and only if there is a substitution θ such that $t = u\theta$. This implies that $t \to_{\mathcal{B}_{E\cup R}} r_k$ if and only if there is an unfrozen context C, ground substitution θ, and rule $l \to r \in \overrightarrow{E} \cup R$ such that $t = C[l\theta]$. From an inductive argument, it also follows that $t \to_{\mathcal{B}_{E\cup R}} c_s$ if and only if $t \in T_{\Sigma,s}$ and $pos_{\overline{\nu}}(t) \subseteq pos_\Upsilon(t)$. Hence, $t \in L(\mathcal{B}_{E\cup R})_k$ if and only if $[t]_B \in \mathit{Norm}_{\mathcal{R}_{E\cup R}/B,k}$ and $[t]_B \notin C^\Upsilon_{\mathcal{R}/B,k}$. The construction and proof for $\mathcal{B}_E$ are entirely similar to those for $\mathcal{B}_{E\cup R}$, with $\mathcal{B}_E = (\Sigma^K, Q, \Gamma_{CSC}, \Delta_E)$, where Δ_E is defined as $\Delta_{E\cup R}$ but omitting the rules R (with the appropriate E-constructors Ω and I_E). By mimicking the argument above, it is straightforward to show that $t \in L(\mathcal{B}_E)_k$ if and only if $[t]_B \in \mathit{Norm}_{\mathcal{R}_E/B,k}$ and $[t]_B \notin T_{\Omega/B,k}$. □

Theorem 4. Let $\mathcal{R} = (\Sigma, E\cup B, R, \varphi)$ be a simple generalized rewrite theory that canonically sufficiently complete relative to the constructor signature pair (Ω,Υ). If $\mathcal{R}_E$ and $\mathcal{R}_{E\cup R}$ are PTA-checkable, then there are PTAs $\mathcal{F}_E$ and $\mathcal{D}_{E\cup R}$ such that Ω is a signature of E-free constructors modulo V if and only if $L(\mathcal{F}_E) = \varnothing$, and Υ is signature of $\mathcal{R}$-deadlock constructors if and only if $L(\mathcal{D}_{E\cup R}) = \varnothing$.

Proof. Let $\Sigma = (S, \leq, F)$, and let $\Sigma^K = (K, F^K)$. Define $I_E = \{t \mid C[t] \in \mathit{lhs}(\overrightarrow{E}) \wedge t \notin X \wedge C \neq \Box\}$, and $I_{E\cup R} = \{t \mid C[t] \in \mathit{lhs}(\overrightarrow{E} \cup R) \wedge t \notin X \wedge C \neq \Box\}$. The construction of $\mathcal{F}_E = (\Sigma^K, Q_E, \Gamma_E, \Delta_E)$ is as follows:

- $Q_E = \{Q_{E,k}\}_{k\in K}$ with $Q_{E,k} = \{r_k^E\} \cup \{c_s \mid s \in S\} \cup \{p_s \mid s \in S\} \cup \{p_u \mid u \in I_E \cap T_{\Sigma,k}\}$,
- $\Gamma_E = \{\gamma_{E,k}\}_{k\in K}$ with $\gamma_{E,k} = r_k^E \wedge \bigvee_{s\in[k]_\leq} p_s \wedge c_s$, and
- $\Delta_E = \{f(p_{s_1},\ldots,p_{s_n}) \to p_s \mid f \in F_{s_1\ldots s_n,s}\}$

$$
\begin{aligned}
&\cup \{c(c_{s_1}, \ldots, c_{s_n}) \to c_s \mid c \in \Omega_{s_1 \ldots s_n, s}\} \\
&\cup \{p_s \to p_{s'} \mid s, s' \in S \wedge s < s'\} \\
&\cup \{c_s \to c_{s'} \mid s, s' \in S \wedge s < s'\} \\
&\cup \{f(p_{t_1}, \ldots, p_{t_n}) \to p_{f(t_1, \ldots, t_n)} \mid f(t_1, \ldots, t_n) \in I_E\} \\
&\cup \{f(p_{t_1}, \ldots, p_{t_n}) \to r_k^E \mid f(t_1, \ldots, t_n) \in T_{\Sigma,k} \cap lhs(\overrightarrow{E})\} \\
&\cup \{f(p_{k_1}, \ldots, r_{k_i}^E, \ldots, p_{k_n}) \to r_k^E \mid f \in F_{k_1 \ldots k_n, k}^K\}.
\end{aligned}
$$

The construction of $\mathcal{D}_{E \cup R} = (\Sigma^K, Q_{E \cup R}, \Gamma_{E \cup R}, \Delta_{E \cup R})$ is as follows:

- $Q_{E \cup R} = \{Q_{E \cup R,k}\}_{k \in K}$ with $Q_{E \cup R,k} = \{r_k^E, r_k^R\} \cup \{c_s \mid s \in S\} \cup \{p_s \mid s \in S\} \cup \{p_u \mid u \in I_{E \cup R} \cap T_{\Sigma,k}\}$,
- $\Gamma_{E \cup R} = \{\gamma_{E \cup R,k}\}_{k \in K}$ with $\gamma_{E \cup R,k} = r_k^R \wedge \neg r_k^E \wedge \bigvee_{s \in [k]_\leq} p_s \wedge c_s$, and
- $\Delta_{E \cup R} = \{f(OBp_{s_1}, \ldots, p_{s_n}) \to p_s \mid f \in F_{s_1 \ldots s_n, s}\}$
  $\cup \{c(\mathrm{rep}_c^1(s_1), \ldots, \mathrm{rep}_c^n(s_n)) \to c_s \mid c \in \Upsilon_{s_1 \ldots s_n, s}\}$
  $\cup \{p_s \to p_{s'} \mid s, s' \in S \wedge s < s'\}$
  $\cup \{c_s \to c_{s'} \mid s, s' \in S \wedge s < s'\}$
  $\cup \{f(p_{t_1}, \ldots, p_{t_n}) \to p_{f(t_1, \ldots, t_n)} \mid f(t_1, \ldots, t_n) \in I_{E \cup R}\}$
  $\cup \{f(p_{t_1}, \ldots, p_{t_n}) \to r_k^E \mid f(t_1, \ldots, t_n) \in T_{\Sigma,k} \cap lhs(\overrightarrow{E})\}$
  $\cup \{f(p_{t_1}, \ldots, p_{t_n}) \to r_k^R \mid f(t_1, \ldots, t_n) \in T_{\Sigma,k} \cap lhs(R)\}$
  $\cup \{f(p_{k_1}, \ldots, r_{k_i}^E, \ldots, p_{k_n}) \to r_k^E \mid f \in F_{k_1 \ldots k_n, k}^K\}$
  $\cup \{f(p_{k_1}, \ldots, r_{k_i}^R, \ldots, p_{k_n}) \to r_k^R \mid f \in F_{k_1 \ldots k_n, k}^K \wedge i \in \overline{\varphi}(f)\}$,
  where $\mathrm{rep}_c^i(s)$ equals c_s if $i \in \overline{\varphi}(c)$, and p_s otherwise.

Let $t \in T_\Sigma$. By structural induction on t, it follows that $t \to_{\mathcal{F}_E} p_s$ (resp. $t \to_{\mathcal{D}_{E \cup R}} p_s$) if and only if $t \in T_{\Sigma,s}$. A similar inductive argument shows that $t \to_{\mathcal{F}_E} p_u$ (resp. $t \to_{\mathcal{D}_{E \cup R}} p_u$) if and only if there is a substitution θ such that $t = u\theta$. This implies that: (i) $t \to_{\mathcal{F}_E} r_k^E$ (resp. $t \to_{\mathcal{D}_{E \cup R}} r_k^E$) if and only if there is a context C, ground substitution θ, and rule $l \to r \in \overrightarrow{E}$ such that $t = C[l\theta]$, and (ii) $t \to_{\mathcal{D}_{E \cup R}} r_k^R$ if and only if there is an unfrozen context C, ground substitution θ, and rule $l \to r \in R$ such that $t = C[l\theta]$. From an inductive argument, it also follows that $t \to_{\mathcal{F}_E} c_s$ (resp. $t \to_{\mathcal{D}_{E \cup R}} c_s$) if and only if $t \in T_{\Omega,s}$ (resp. $t \in T_{\Sigma,s}$ and $pos_{\overline{\varphi}}(t) \subseteq pos_\Upsilon(t)$). Hence, $t \in L(\mathcal{F}_E)_k$ if and only if $[t]_B \in T_{\Omega,E/B,k}$ and $[t]_B \notin Norm_{\mathcal{R}_E/B,k}$, and $t \in L(\mathcal{D}_{E \cup R})_k$ if and only if $[t]_B \in C_{\mathcal{R}/B,k}^\Upsilon$ and $[t]_B \notin Norm_{\mathcal{R}_{E \cup R}/B,k}$. Observe that because $\mathcal{R}$ is a simple generalized rewrite theory, it is correct to ignore frozen contexts in the automata $\mathcal{F}_E$ and $\mathcal{D}_{E \cup R}$ when only considering equations. $\square$

A.2 Mechanical Proofs

In this section, the mechanical proofs of ground sort-decreasingness, confluence, and coherence are obtained with the Maude Church-Rosser Checker and the Maude Coherence Checker tools [34]. The proofs of ground operational termination are obtained with the Maude Termination Tool [32]. Inductive claims about initial algebras are proved using the current version of Maude's Inductive Theorem Prover [52].

A.2.1 Proofs for `BAG-CHOICE+CARD`

The input specification is

$$(\Sigma^{\texttt{BCC}}, E^{\texttt{BCC}} \cup B^{\texttt{BCC}}, \mu^{\texttt{BCC}}, R^{\texttt{BCC}}, \nu^{\texttt{BCC}})$$

as in Section 3.3. In order to simplify notation, the expression `BCC` denotes `BAG-CHOICE+CARD`, $\texttt{BCC}_E$ denotes $\texttt{BCC}_{E^{\texttt{BCC}}}$, and $\texttt{BCC}_{E \cup R}$ denotes $\texttt{BCC}_{E^{\texttt{BCC}} \cup R^{\texttt{BCC}}}$.

Lemma 8. `BCC` *is admissible.*

Proof.

- The rewrite rules in $\texttt{BCC}_E$ are ground sort-decreasing and confluent modulo $B^{\texttt{BCC}}$:

```
Maude> (check Church-Rosser .)
rewrites: 26197 in 76ms cpu (83ms real) (341226 rewrites/second)

Church-Rosser checking of BAG-CHOICE+CARD
Checking solution:
The following critical pairs cannot be joined:
  cp s | #4:Bag |
    = s | mt #4:Bag | .
The specification is sort-decreasing.
```

  Despite the fact that this critical pair is not joinable automatically by the CRC tool, it is easy to see that it is indeed joinable because `mt` is the identity operator for `__` and the structural axioms $B^{\texttt{BCC}}$ are not affected by the strategy map $\mu^{\texttt{BCC}}$.

- The rewrite rules in $\texttt{BCC}_E$ are ground operationally terminating modulo $B^{\texttt{BCC}}$: this proof is obtained with the MTT using AProVE as the backend. The proof is rather long, so a reduced snapshot of it is presented.

```
START Maude C;Uk;B false false
SUCCESSFULLY 4 Maude seconds 3

(VAR B N V1 V2 V X X@@@)
(THEORY
 (AC ---osb-Bag-csb-osb-Bag-csb)
)
...
Using the Dependency Graph resulted in no new DP problems.

Termination of R successfully shown.

Duration:
0:00 minutes
```

- $R^{\mathtt{BCC}}$ and $E^{\mathtt{BCC}}$ are ground-coherent modulo $B^{\mathtt{BCC}}$:

```
Maude> (check coherence .)
rewrites: 24006 in 71ms cpu (75ms real) (336378 rewrites/second)
Coherence checking of BAG-CHOICE+CARD
Coherence checking solution:
All critical pairs have been rewritten and all equations are non-constructor.
The specification is ground coherent.
```

□

Lemma 9. $\mathbf{Can}_{\Sigma^{\mathtt{BCC}},E^{\mathtt{BCC}}/B^{\mathtt{BCC}}} \cong \mathbf{Can}^{\mu^{\mathtt{BCC}}}_{\Sigma^{\mathtt{BCC}},E^{\mathtt{BCC}}/B^{\mathtt{BCC}}}$.

Proof. The isomorphism is checked with the SCC's command "`ccc`" (see [54] for details):

```
Maude> (ccc BAG-CHOICE+CARD .)
Checking canonical completeness of BAG-CHOICE+CARD ...
Success: BAG-CHOICE+CARD is canonically complete.
```

□

Lemma 10. *The rewrite rules in* $\mathtt{BCC}_{E\cup R}$ *are ground sort-decreasing and ground weakly-normalizing modulo* $B^{\mathtt{BCC}}$.

Proof.

- The rewrite rules in $\mathtt{BCC}_{E\cup R}$ are ground sort-decreasing modulo $B^{\mathtt{BCC}}$:

```
Maude> (check Church-Rosser .)
rewrites: 26197 in 76ms cpu (83ms real) (341226 rewrites/second)

Church-Rosser checking of hatBAG-CHOICE+CARD
Checking solution:
...
The specification is sort-decreasing.
```

- The rewrite rules in BCC are ground weakly-normalizing modulo $B^{\mathtt{BCC}}$: this proof is obtained with the MTT using AProVE as the backend. The proof is rather long, so a reduced snapshot of it is presented.

```
START Maude C;Uk;B false false
SUCCESSFULLY 5 Maude seconds 3

(VAR B N NeB V1 V2 V X X@@@)
(THEORY
 (AC ---osb-Bag-csb-osb-Bag-csb)
)
...
Using the Dependency Graph resulted in no new DP problems.

Termination of R successfully shown.

Duration:
0:00 minutes
```

□

A.2.2 Proofs for CHANNEL

The input specification is

$$(\Sigma^{\mathtt{CHANNEL}}, E^{\mathtt{CHANNEL}} \cup B^{\mathtt{CHANNEL}}, R^{\mathtt{CHANNEL}})$$

as in Section 3.6. In order to simplify notation, the expression $\mathtt{CHANNEL}_E$ denotes $\mathtt{CHANNEL}_{E^{\mathtt{CHANNEL}}}$ and $\mathtt{CHANNEL}_{E \cup R}$ denotes $\mathtt{CHANNEL}_{E^{\mathtt{CHANNEL}} \cup R^{\mathtt{CHANNEL}}}$.

Lemma 11. CHANNEL *is admissible.*

Proof.

- The rewrite rules in $\mathtt{CHANNEL}_E$ are ground local confluent and sort-decreasing modulo $B^{\mathtt{CHANNEL}}$.

```
Maude> (check Church-Rosser .)
rewrites: 9474 in 34ms cpu (37ms real) (278606 rewrites/second)
Church-Rosser checking of CHANNEL
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
The specification is sort-decreasing.
```

- The rewrite rules in $\mathtt{CHANNEL}_E$ are ground operationally-terminating: this proof is obtained with the MTT using μ-Term as the backend:

```
YES
```

```
Problem 1:

(VAR M N L L')
(RULES
_@_(_;_(N,L),L') -> _;_(N,_@_(L,L'))
_@_(nil,L) -> L
)

Problem 1:

Order-Sorted Dependency Pairs Processor:
-> Pairs:
_@_#(_;_(N,L),L') -> _@_#(L,L')
-> Rules:
_@_(_;_(N,L),L') -> _;_(N,_@_(L,L'))
_@_(nil,L) -> L

Problem 1:

SCC Processor:
-> Pairs:
_@_#(_;_(N,L),L') -> _@_#(L,L')
-> Rules:
_@_(_;_(N,L),L') -> _;_(N,_@_(L,L'))
_@_(nil,L) -> L
->Strongly Connected Components:
->->Cycle:
->->-> Pairs:
_@_#(_;_(N,L),L') -> _@_#(L,L')
->->-> Rules:
_@_(_;_(N,L),L') -> _;_(N,_@_(L,L'))
_@_(nil,L) -> L

Problem 1:

SubNColl Processor:
-> Pairs:
_@_#(_;_(N,L),L') -> _@_#(L,L')
-> Rules:
_@_(_;_(N,L),L') -> _;_(N,_@_(L,L'))
_@_(nil,L) -> L
->Projection:
pi(_@_#) = 1

Problem 1:

SCC Processor:
-> Pairs:
Empty
-> Rules:
_@_(_;_(N,L),L') -> _;_(N,_@_(L,L'))
_@_(nil,L) -> L
->Strongly Connected Components:
There is no strongly connected component
```

```
The problem is finite.
```

- $R^{\texttt{CHANNEL}}$ and $E^{\texttt{CHANNEL}}$ are ground-coherent modulo $B^{\texttt{CHANNEL}}$:

```
Maude> (check coherence .)
rewrites: 11760 in 37ms cpu (43ms real) (315408 rewrites/second)
Coherence checking of CHANNEL
Coherence checking solution:
All critical pairs have been rewritten and all equations are non-constructor.
The specification is coherent.
```

□

Lemma 12. *The rewrite rules in* $\texttt{CHANNEL}_{E \cup R}$ *are ground sort-decreasing, confluent, and operationally terminating modulo* $B^{\texttt{CHANNEL}}$.

Proof.

- The rewrite rules in $\texttt{CHANNEL}_{E \cup R}$ are ground sort-decreasing and local confluent modulo $B^{\texttt{CHANNEL}}$:

```
Maude> (check Church-Rosser hatCHANNEL .)
rewrites: 51279 in 141ms cpu (157ms real) (361845 rewrites/second)
Church-Rosser checking of hatCHANNEL
Checking solution:
All critical pairs have been joined.
The specification is locally-confluent.
The specification is sort-decreasing.
```

- The rewrite rules in $\texttt{CHANNEL}_{E \cup R}$ are ground strongly-normalizing: this proof is obtained with the MTT using AProVE as the backend. The proof is rather long, so a reduced snapshot of it is presented. Observe that this proof implies the ground strong-normalization of $\texttt{CHANNEL}_{E}$.

```
START AProVE
SUCCESSFULLY AProVE seconds: 3
...
   R     ->Dependency Pair Analysis
...
       ->DP Problem 1
         ->SCP
       ->DP Problem 2
         ->SCP
       ->DP Problem 3
         ->SCP
       ->DP Problem 4
         ->Polo
           ->DP Problem 5
             ->DGraph
             ...
               ->DP Problem 6
                 ->Polynomial Ordering
...
```

```
Using the Dependency Graph resulted in no new DP problems.
Termination of R successfully shown.
Duration:
0:01 minutes
```

□

Lemma 13. *The following are inductive consequences of* $\mathcal{E}_{\texttt{CHANNEL}}$*:*

1. `nil` *is the right identity of* `_@_`*, and*

2. `_@_` *is associative.*

Proof. In order to avoid name clashes between the names in Maude's prelude, such as `Nat` and `List`, the sorts `Nat` and `List` in `CHANNEL` are renamed to `Nat2` and `List2`, respectively. Both goals are discharged automatically by structural induction on the equational constructors $\Omega_{\texttt{CHANNEL,List2}}$ via the ITP's command `ind*` which creates constructor-based structural induction goals and tries to discharge them automatically. For a detailed explanation of the ITP's commands see [52].

1. $\mathcal{E}_{\texttt{CHANNEL}} \models (\forall l : \texttt{List})\, l \,\texttt{@}\, \texttt{nil} = l$

```
==================================
label-sel: id@0
==================================
A{l:List2} l:List2 @ nil = l:List2
++++++++++++++++++++++++++++++++++
(ind* on l:List2 .)
rewrites: 625 in 4ms cpu (4ms real) (141019 rewrites/second)
Eliminated current goal.
q.e.d
```

2. $\mathcal{E}_{\texttt{CHANNEL}} \models (\forall l, l', l'' : \texttt{List})\, l \,\texttt{@}\;\; (l' \,\texttt{@}\, l'') = (l \,\texttt{@}\, l') \,\texttt{@}\, l''$

```
==================================
label-sel: assoc@0
==================================
A{l''':List2 ; l':List2 ; l:List2}
 l:List2 @(l':List2 @ l''':List2) = (l:List2 @ l':List2)@ l''':List2
++++++++++++++++++++++++++++++++++
(ind* on l:List2 .)
rewrites: 1126 in 5ms cpu (5ms real) (211098 rewrites/second)
Eliminated current goal.
q.e.d
```

□

APPENDIX B

MISSING PROOFS FOR CHAPTER 5

This appendix includes the ABP specification and predicates in Maude, proofs of admissibility, and the missing proofs in the formal verification of the protocol in Section 5.2.

The ABP specification can be found in `abp.maude` and the predicate specification in `abp.preds.maude`, both available for download with this dissertation. The mechanical proofs were obtained with the tools integrated in the current version of the Maude Formal Environment (MFE) [36, 35].

B.1 `abp.maude`

This file contains the Maude specification of the ABP.

```
--- Booleans
fmod IBOOL is
  --- use Maude built-in Booleans
  pr BOOL-OPS .
endfm

---- Natural numbers
fmod INAT is
  pr IBOOL .

  sort iNat .
  op 0 : -> iNat [ctor] .
  op s_ : iNat -> iNat [ctor] .

  vars N N' : iNat .

  --- equality enrichment
  op _~_ : iNat iNat -> Bool [comm] .
  eq N ~ N
   = true .
```

```
  eq s N ~ s N'
   = N ~ N' .
  eq 0 ~ s N
   = false .
  eq N ~ s N
   = false .
endfm

--- bits
fmod BIT is
  pr IBOOL .
  sort Bit .

  ops on off : -> Bit [ctor] .
  op flip : Bit -> Bit .
  eq flip(on) = off .
  eq flip(off) = on .

  --- equality enrichment
  op _~_ : Bit Bit -> Bool [comm] .
  eq B:Bit ~ B:Bit
   = true .
  eq on ~ off
   = false .
  eq B:Bit ~ flip(B:Bit)
   = false .
endfm

--- list of naturals
fmod INAT-LIST is
  pr INAT .
  sort iNatList .
  op nil : -> iNatList [ctor] .
  op __ : iNat iNatList -> iNatList [ctor prec 61] .

  vars N N'   : iNat .
  vars NL NL' : iNatList .

  op _;_ : iNatList iNatList -> iNatList [prec 65] .
  eq nil ; NL = NL .
  eq N NL ; NL' = N (NL ; NL') .

  --- equality enrichment
  op _~_ : iNatList iNatList -> Bool [comm] .
  eq nil ~ (N NL)
   = false .
  eq NL ~ NL
   = true .
  eq (N NL) ~ (N' NL')
   = (N ~ N') and (NL ~ NL') .
endfm

--- queue of bits
```

```
fmod BIT-QUEUE is
  pr BIT .
  sort BitQueue .
  op nil : -> BitQueue [ctor] .
  op __ : Bit BitQueue -> BitQueue [ctor prec 61] .

  vars B B'   : Bit .
  vars BQ BQ' : BitQueue .

  op _;_ : BitQueue BitQueue -> BitQueue [prec 65] .
  eq nil ; BQ = BQ .
  eq B BQ ; BQ' = B (BQ ; BQ') .
endfm

--- packets: pair of bit and nat
fmod BIT-PACKET is
  pr BIT .
  pr INAT .
  sort BitPacket .
  op `(_,_`) : Bit iNat -> BitPacket [ctor] .

  vars B B'   : Bit .
  var  BP     : BitPacket .
  vars N N'   : iNat .

  --- equality enrichment
  op _~_ : BitPacket BitPacket -> Bool [comm] .
  eq BP ~ BP
   = true .
  eq (B,N) ~ (B',N')
   = B ~ B' and N ~ N' .
endfm

--- queue of pairs of bits and nats
fmod BIT-PACKET-QUEUE is
  pr BIT-PACKET .
  sort BitPacketQueue .
  op nil : -> BitPacketQueue [ctor] .
  op __ : BitPacket BitPacketQueue -> BitPacketQueue [ctor prec 61] .

  vars B B'     : Bit .
  vars BP BP'   : BitPacket .
  vars BPQ BPQ' : BitPacketQueue .
  var  N        : iNat .

  op _;_ : BitPacketQueue BitPacketQueue -> BitPacketQueue [prec 65] .
  eq nil ; BPQ = BPQ .
  eq BP BPQ ; BPQ' = BP (BPQ ; BPQ') .
endfm

--- state syntax
fmod ABP-STATE is
  pr BIT-PACKET-QUEUE .
```

```
  pr BIT-QUEUE .
  pr INAT-LIST .

  sort Sys .
  op _:_>_|_<_:_ : iNat Bit BitPacketQueue
                   BitQueue Bit iNatList -> Sys [ctor] .
endfm

--- transitions
mod ABP is
  pr ABP-STATE .

  vars B B' B''  : Bit .
  var  BP        : BitPacket .
  vars BQ BQ'    : BitQueue .
  vars BPQ BPQ'  : BitPacketQueue .
  vars N N'      : iNat .
  vars NL NL'    : iNatList .

  rl [send-1] :
     N : B > BPQ | BQ < B' : NL
  => N : B > BPQ ; ((B, N) nil) | BQ < B' : NL .

  rl [recv-1a] :
     N : B > BPQ | B BQ < B' : NL
  => N : B > BPQ | BQ < B' : NL .

  rl [recv-1b] :
     N : on > BPQ | off BQ < B' : NL
  => s(N) : off > BPQ | BQ < B' : NL .

  rl [recv-1c] :
     N : off > BPQ | on BQ < B' : NL
  => s(N) : on > BPQ | BQ < B' : NL .

  rl [send-2] :
     N : B > BPQ | BQ < B' : NL
  => N : B > BPQ | BQ ; (B' nil) < B' : NL .

  rl [recv-2a] :
     N : B > (on,N') BPQ | BQ < on : NL
  => N : B > BPQ | BQ < off : (N' NL) .

  rl [recv-2b] :
     N : B > (off,N') BPQ | BQ < off : NL
  => N : B > BPQ | BQ < on : (N' NL) .

  rl [recv-2c] :
     N : B > (off,N') BPQ | BQ < on : NL
  => N : B > BPQ | BQ < on : NL .

  rl [recv-2d] :
     N : B > (on,N') BPQ | BQ < off : NL
```

```
  => N : B > BPQ | BQ < off : NL .

  rl [drop-1a] :
     N : B > (off,N') BPQ | BQ < B' : NL
  => N : B > BPQ | BQ < B' : NL .

  rl [drop-1b] :
     N : B > (on,N') BPQ | BQ < B' : NL
  => N : B > BPQ | BQ < B' : NL .

  rl [dup-1] :
     N : B > BP BPQ | BQ < B' : NL
  => N : B > BP (BP BPQ) | BQ < B' : NL .

  rl [drop-2a] :
     N : B > BPQ | off BQ < B' : NL
  => N : B > BPQ | BQ < B' : NL .

  rl [drop-2b] :
     N : B > BPQ | on BQ < B' : NL
  => N : B > BPQ | BQ < B' : NL .

  rl [dup-2] :
     N : B > BPQ | B'' BQ < B' : NL
  => N : B > BPQ | B'' (B'' BQ) < B' : NL .
endm
```

B.2 ABP Admissibility and Free Constructors Modulo

This section presents the mechanical proofs for the admissibility of module ABP and for the equational freeness of its subsignature of constructors.

For ground sort-decreasingness, operational termination, confluence, and coherence, the following is the output of the mechanical proof:

```
Maude> (ccr ABP .)
rewrites: 8114680 in 5145ms cpu (5146ms real) (1577130 rewrites/second)
Church-Rosser check for ABP
  All critical pairs have been joined.
  The specification is locally-confluent.
  The module is sort-decreasing.

Maude> (ctf ABP .)
rewrites: 95604 in 141ms cpu (2796ms real) (673371 rewrites/second)
Success: The functional part of module ABP is terminating.

Maude> (cch ABP .)
rewrites: 2188028 in 1470ms cpu (1470ms real) (1487669 rewrites/second)
Coherence checking of ABP
  All critical pairs have been rewritten and no rewrite with rules can
  happen at non-overlapping positions of equations left-hand sides.
```

For sufficient completeness and equational freeness of constructors modulo, the following is the output of the mechanical proof:

```
Maude> (scc ABP-STATE .)
rewrites: 3285 in 7ms cpu (8ms real) (410676 rewrites/second)
Sufficient completeness check for ABP-STATE
  Completeness counter-examples: none were found
  Freeness counter-examples: none were found
```

B.3 abp.preds.maude

This section contains the Maude specification of the state predicates, with the additional lemmata.

```
----- predicate defining initial states
fmod ABP-PRED-INIT is
  pr ABP-STATE .

  --- initial states
  op init : Sys -> [Bool] .
  eq [init-1] :
     init( 0 : on > nil | nil < on : nil)
   = true .
  eq [init-2] :
     init( 0 : off > nil | nil < off : nil)
   = true .
endfm

---- strengthening
fmod ABP-PRED-GOOD-QUEUES is
  pr ABP-STATE .

  vars B1 B2 B     : Bit .
  var  BP          : BitPacket .
  var  BPQ         : BitPacketQueue .
  var  BQ          : BitQueue .
  vars N N'        : iNat .
  var  NL          : iNatList .

  op good-queues : Sys -> Bool .
  eq [good-queues-1a] :
     good-queues(N : on > BPQ | BQ < on : NL)
   = all-bits(BQ,on) and good-packet-queue(BPQ,on,N) .
  eq [good-queues-1b] :
     good-queues(N : off > BPQ | BQ < off : NL)
   = all-bits(BQ,off) and good-packet-queue(BPQ,off,N) .
  eq [good-queues-2a] :
     good-queues(N : on > BPQ | BQ < off : NL)
   = good-bit-queue(BQ,off) and all-packets(BPQ,on,N) .
  eq [good-queues-2b] :
     good-queues(N : off > BPQ | BQ < on : NL)
```

```
   = good-bit-queue(BQ,on) and all-packets(BPQ,off,N) .

  ------------------------------------------------
  --- auxiliary functions for the queue of bits ---
  ------------------------------------------------
  op good-bit-queue : BitQueue Bit -> Bool .
  eq [gbq-1] :
     good-bit-queue(nil,B)
   = true .
 ceq [gbq-2] :
     good-bit-queue(B1 BQ, B)
   = good-bit-queue(BQ,B)
  if B1 = flip(B) .
  eq [gbq-3] :
     good-bit-queue(B BQ, B)
   = all-bits(BQ,B) .

  op all-bits : BitQueue Bit -> Bool .
  eq [ab-1] :
     all-bits((nil).BitQueue,B)
   = true .
  eq [ab-2] :
     all-bits(B1 BQ,B)
   = B1 ~ B and all-bits(BQ,B) .

  ------------------------------------------------------
  --- auxiliary functions for the queue of packets ---
  ------------------------------------------------------
  op good-packet-queue : BitPacketQueue Bit iNat -> Bool .
  eq [gpq-1] :
     good-packet-queue(nil,B,N)
   = true .
 ceq [gpq-2] :
     good-packet-queue((B1,N') BPQ,B,N)
   = N ~ s(N') and good-packet-queue(BPQ,B,N)
  if B1 = flip(B) .
  eq [gpq-3] :
     good-packet-queue((B,N') BPQ,B,N)
   = N ~ N' and all-packets(BPQ,B,N) .

  op all-packets : BitPacketQueue Bit iNat -> Bool .
  eq [ap-1] :
     all-packets((nil).BitPacketQueue,B,N)
   = true .
  eq [ap-2] :
     all-packets(BP BPQ,B,N)
   = BP ~ (B,N) and all-packets(BPQ,B,N) .
endfm

---- lemmata for strengthening
fmod ABP-PRED-GOOD-QUEUES-LEMMATA is
  pr ABP-PRED-GOOD-QUEUES .
```

```
  vars B1 B2 B     : Bit .
  var  BP          : BitPacket .
  var  BPQ         : BitPacketQueue .
  var  BQ          : BitQueue .
  vars N N'        : iNat .
  var  NL          : iNatList .

  ------------------------
  --- auxiliary lemmas ---
  ------------------------

  --- queues of bits
  eq [lem-bq1] :
     good-bit-queue(B B BQ,B1)
   = good-bit-queue(B BQ,B1) .
  eq [lem-bq2] :
     all-bits(BQ ; (B nil),B)
   = all-bits(BQ,B) .
  eq [lem-bq3] :
     good-bit-queue(BQ ; (B nil),B)
   = good-bit-queue(BQ,B) .
 ceq [lem-bq4] :
     good-bit-queue(BQ,B)
   = true
  if all-bits(BQ,B) = true .
 ceq [lem-bq5] :
     good-bit-queue(BQ,B)
   = true
  if all-bits(BQ,flip(B)) = true .

  --- queues of packets
  eq [lem-pq1] :
     good-packet-queue(BP BP BPQ,B,N)
   = good-packet-queue(BP BPQ,B,N) .
  eq [lem-pq2] :
     all-packets(BPQ ; (B,N) nil,B,N)
   = all-packets(BPQ,B,N) .
  eq [lem-pq3] :
     good-packet-queue(BPQ ; (B,N) nil,B,N)
   = good-packet-queue(BPQ,B,N) .
 ceq [lem-pq4] :
     good-packet-queue(BPQ,B,N)
   = true
  if all-packets(BPQ,B,N) .
 ceq [lem-pq5] :
     good-packet-queue(BPQ,B,s(N))
   = true
  if all-packets(BPQ,flip(B),N) = true .
endfm

---- main invariant
fmod ABP-PRED-INV is
  pr ABP-STATE .
```

```
  vars B1 B2 B     : Bit .
  var  BPQ         : BitPacketQueue .
  var  BQ          : BitQueue .
  vars N N'        : iNat .
  var  NL          : iNatList .

  --- main invariant
  op inv : Sys -> Bool .
  eq [inv-1a] :
     inv(N : on > BPQ | BQ < on : NL)
   = (N NL) ~ gen-list(N) .
  eq [inv-1a] :
     inv(N : off > BPQ | BQ < off : NL)
   = (N NL) ~ gen-list(N) .
  eq [inv-2a] :
     inv(N : on > BPQ | BQ < off : NL)
   = NL ~ gen-list(N) .
  eq [inv-2a] :
     inv(N : off > BPQ | BQ < on : NL)
   = NL ~ gen-list(N) .

  --- less fine-grained invariant
  op inv-main : Sys -> Bool .
  eq [inv-main-1] :
     inv-main(N : B > BPQ | BQ < B : NL)
   = (N NL) ~ gen-list(N) .
 ceq [inv-main-2] :
     inv-main(N : B1 > BPQ | BQ < B2 : NL)
   = NL ~ gen-list(N)
  if B1 ~ B2 = false .
  ------------------------------------
  --- auxiliary generation of lists ---
  ------------------------------------
  op gen-list : iNat -> iNatList .
  eq gen-list(0)
  = (0 nil) .
  eq gen-list(s N)
   = (s N) gen-list(N) .
endfm

---- all predicates
fmod ABP-PREDS is
  pr ABP-PRED-INIT .
  pr ABP-PRED-GOOD-QUEUES .
  pr ABP-PRED-INV .
endfm

---- all predicates and lemmata
fmod ABP-PREDS+LEMMATA is
  pr ABP-PREDS .
  pr ABP-PRED-GOOD-QUEUES-LEMMATA .
endfm
```

B.4 ABP-PREDS is Admissible

This section presents the mechanical proofs for the admissibility of module ABP-PREDS.

For ground sort-decreasingness, operational termination, confluence, and coherence, the following is the output of the mechanical proofs:

```
Maude> (ccr ABP-PREDS .)
rewrites: 13992680 in 9647ms cpu (9649ms real) (1450389 rewrites/second)
Church-Rosser check for ABP-PREDS
The following critical pairs must be proved joinable:
  ccp ABP-PREDS1614 for gbq-3 and gbq-2
    all-bits(BQ:BitQueue,B1:Bit)
    = good-bit-queue(BQ:BitQueue,B1:Bit)
    if B1:Bit = flip(B1:Bit).
  ccp ABP-PREDS1617 for gpq-3 and gpq-2
    N:iNat ~ N':iNat and all-packets(BPQ:BitPacketQueue,B1:Bit,N:iNat)
    = N:iNat ~ s N':iNat and
     good-packet-queue(BPQ:BitPacketQueue,B1:Bit,N:iNat)
    if B1:Bit = flip(B1:Bit).
    The module is sort-decreasing.

Maude> (ctf ABP-PREDS .)
rewrites: 209594 in 378ms cpu (3363ms real) (553101 rewrites/second)
Success: The functional part of module ABP-PREDS is terminating.

Maude> (cch ABP-PREDS .)
rewrites: 157674 in 227ms cpu (227ms real) (691658 rewrites/second)
Coherence checking of ABP-PREDS
  All critical pairs have been rewritten and no rewrite with rules can happen
  at non-overlapping positions of equations left-hand sides.
```

The ground confluence check for ABP-PREDS returns two critical pairs. These critical pairs are joinable because their conditions are trivially unfeasible. Therefore ABP-PREDS is admissible.

B.5 ABP-PREDS+LEMMATA is Admissible

This section presents the mechanical proofs for the admissibility of module ABP-PREDS+LEMMATA.

For ground sort-decreasingness, operational termination, confluence, and coherence, the following is the output of the mechanical proofs:

```
Maude> (ccr ABP-PREDS+LEMMATA .)
rewrites: 19037580 in 40313ms cpu (42315ms real) (472234 rewrites/second)
Church-Rosser check for ABP-PREDS+LEMMATA
The following critical pairs must be proved joinable:
  cp ABP-PREDS+LEMMATA12
    good-packet-queue(#1:BitPacket(#2:BitPacketQueue ;
```

```
                    (B:Bit,N:iNat)nil),B:Bit,N:iNat)
  = good-packet-queue(#1:BitPacket #2:BitPacketQueue,B:Bit,N:iNat).
cp ABP-PREDS+LEMMATA9
  good-bit-queue(#1:Bit(#2:BitQueue ; B:Bit nil),B:Bit)
  = good-bit-queue(#1:Bit #2:BitQueue,B:Bit).
ccp ABP-PREDS+LEMMATA1632 for lem-bq3 and lem-bq4
  good-bit-queue(#1:BitQueue,B:Bit)
  = true
  if all-bits(#1:BitQueue ; B:Bit nil,B:Bit)= true .
ccp ABP-PREDS+LEMMATA1633 for lem-bq3 and lem-bq5
  good-bit-queue(#1:BitQueue,B:Bit)
  = true
  if all-bits(#1:BitQueue ; B:Bit nil,flip(B:Bit))= true .
ccp ABP-PREDS+LEMMATA1636 for gbq-3 and lem-bq4
  all-bits(#2:BitQueue,B:Bit)
  = true
  if all-bits(B:Bit #2:BitQueue,B:Bit)= true .
ccp ABP-PREDS+LEMMATA1638 for gbq-3 and gbq-2
  all-bits(BQ:BitQueue,B1:Bit)
  = good-bit-queue(BQ:BitQueue,B1:Bit)
  if B1:Bit = flip(B1:Bit).
ccp ABP-PREDS+LEMMATA1641 for lem-bq1 and lem-bq4
  good-bit-queue(#1:Bit #2:BitQueue,B:Bit)
  = true
  if all-bits(#1:Bit #1:Bit #2:BitQueue,B:Bit)= true .
ccp ABP-PREDS+LEMMATA1648 for lem-pq3 and lem-pq4
  good-packet-queue(#1:BitPacketQueue,B:Bit,N:iNat)
  = true
  if all-packets(#1:BitPacketQueue ;(B:Bit,N:iNat)nil,B:Bit,N:iNat)= true .
ccp ABP-PREDS+LEMMATA1649 for lem-pq3 and lem-pq5
  good-packet-queue(#1:BitPacketQueue,B:Bit,s N:iNat)
  = true
  if all-packets(#1:BitPacketQueue ;
       (B:Bit,s N:iNat)nil,flip(B:Bit),N:iNat)
  = true .
ccp ABP-PREDS+LEMMATA1652 for lem-pq1 and lem-pq4
  good-packet-queue(#1:BitPacket #2:BitPacketQueue,B:Bit,N:iNat)
  = true
  if all-packets(#1:BitPacket #1:BitPacket #2:BitPacketQueue,B:Bit,N:iNat)
     = true .
ccp ABP-PREDS+LEMMATA1657 for gpq-3 and lem-pq4
  N:iNat ~ #2:iNat and all-packets(#3:BitPacketQueue,B:Bit,N:iNat)
  = true
  if all-packets((B:Bit,#2:iNat)#3:BitPacketQueue,B:Bit,N:iNat)= true .
ccp ABP-PREDS+LEMMATA1659 for gpq-3 and gpq-2
  N:iNat ~ N':iNat and all-packets(BPQ:BitPacketQueue,B1:Bit,N:iNat)
  = N:iNat ~ s N':iNat and good-packet-queue(BPQ:BitPacketQueue,B1:Bit,N:iNat)
  if B1:Bit = flip(B1:Bit).
ccp ABP-PREDS+LEMMATA1679 for lem-bq4 and gbq-2
  true
  = good-bit-queue(BQ:BitQueue,B:Bit)
  if B1:Bit = flip(B:Bit)/\ all-bits(B1:Bit BQ:BitQueue,B:Bit)= true .
ccp ABP-PREDS+LEMMATA1689 for gbq-2 and lem-bq4
  good-bit-queue(#2:BitQueue,B:Bit)
  = true
```

```
    if all-bits(#1:Bit #2:BitQueue,B:Bit)= true /\ #1:Bit = flip(B:Bit).
  ccp ABP-PREDS+LEMMATA1698 for lem-pq4 and gpq-2
    true
    = N:iNat ~ s N':iNat and good-packet-queue(BPQ:BitPacketQueue,B:Bit,N:iNat)
    if B1:Bit = flip(B:Bit)
    /\ all-packets((B1:Bit,N':iNat)BPQ:BitPacketQueue,B:Bit,N:iNat)= true .
  ccp ABP-PREDS+LEMMATA1705 for lem-pq5 and gpq-2
    true
    = N':iNat ~ #3:iNat and good-packet-queue(BPQ:BitPacketQueue,B:Bit,s #3:iNat)
    if B1:Bit = flip(B:Bit)
    /\ all-packets((B1:Bit,N':iNat)BPQ:BitPacketQueue,flip(B:Bit),#3:iNat)= true .
  ccp ABP-PREDS+LEMMATA1708 for gpq-2 and lem-pq4
    N:iNat ~ s #2:iNat and good-packet-queue(#3:BitPacketQueue,B:Bit,N:iNat)
    = true
    if all-packets((#1:Bit,#2:iNat)#3:BitPacketQueue,B:Bit,N:iNat)= true
    /\ #1:Bit = flip(B:Bit).
  ccp ABP-PREDS+LEMMATA1709 for gpq-2 and lem-pq5
    N:iNat ~ #2:iNat and good-packet-queue(#3:BitPacketQueue,B:Bit,s N:iNat)
    = true
    if all-packets((#1:Bit,#2:iNat)#3:BitPacketQueue,flip(B:Bit),N:iNat)= true
    /\ #1:Bit = flip(B:Bit).
    The module is sort-decreasing.

Maude> (ctf ABP-PREDS+LEMMATA .)
rewrites: 258798 in 493ms cpu (3631ms real) (523962 rewrites/second)
Success: The functional part of module ABP-PREDS+LEMMATA is terminating.

Maude> (cch ABP-PREDS+LEMMATA .)
rewrites: 205927 in 291ms cpu (291ms real) (705335 rewrites/second)
Coherence checking of ABP-PREDS+LEMMATA
  All critical pairs have been rewritten and no rewrite with rules can happen
  at non-overlapping positions of equations left-hand sides.
```

The ground confluence check returns 18 critical pairs. These critical pairs can be shown joinable by the *admissible* `ABP-PREDS` module, that is, these proof obligations are inductive equational properties of `ABP-PREDS`. The corresponding ITP proof script is shown below:

```
---(
  cp ABP-PREDS+LEMMATA12
    good-packet-queue(#1:BitPacket(#2:BitPacketQueue ;(B:Bit,N:iNat)nil),B:Bit,N:iNat)
    = good-packet-queue(#1:BitPacket #2:BitPacketQueue,B:Bit,N:iNat).
)
---- assume lemmas lem-pq2 and lem-pq3 previously proved in abp.lemmata.itp
(goal ABP-PREDS+LEMMATA12 : ABP-PREDS |-
  A{ #1:BitPacket ; #2:BitPacketQueue ; B:Bit ; N:iNat }
  (
   (A{ BPQ:BitPacketQueue ; B:Bit ; N:iNat }
     ((all-packets(BPQ ; (B,N) nil,B,N)) = (all-packets(BPQ,B,N)))) &
   (A{ BPQ:BitPacketQueue ; B:Bit ; N:iNat }
     (((good-packet-queue(BPQ ; (B,N) nil,B,N)) = (good-packet-queue(BPQ,B,N)))))
    =>
    (good-packet-queue(
       #1:BitPacket(#2:BitPacketQueue ;(B:Bit,N:iNat)nil),B:Bit,N:iNat))
```

```
     = (good-packet-queue(#1:BitPacket #2:BitPacketQueue,B:Bit,N:iNat))
  )
.)
(cov on good-packet-queue(#1:BitPacket #2:BitPacketQueue,B:Bit,N:iNat) .)
  (ind on V0#1:Bit .)
    (ind* on V0#3:Bit .)
    (ind* on V0#3:Bit .)

---(
  cp ABP-PREDS+LEMMATA9
    good-bit-queue(#1:Bit(#2:BitQueue ; B:Bit nil),B:Bit)
    = good-bit-queue(#1:Bit #2:BitQueue,B:Bit).
)
---- assume lemmas lem-bq2 and lem-bq3 previously proved in abp.lemmata.itp
(goal ABP-PREDS+LEMMATA9 : ABP-PREDS |-
  A{ #1:Bit ; #2:BitQueue ; B:Bit }
  (
   (A{ BQ:BitQueue ; B:Bit } ((all-bits(BQ ; (B nil),B)) = (all-bits(BQ,B)))) &
   (A{ BQ:BitQueue ; B:Bit }
    (((good-bit-queue(BQ ; (B nil),B)) = (good-bit-queue(BQ,B)))))
    =>
    (good-bit-queue(#1:Bit(#2:BitQueue ; B:Bit nil),B:Bit))
     = (good-bit-queue(#1:Bit #2:BitQueue,B:Bit))
  )
.)
(ind on B:Bit .)
  (ind* on #1:Bit .)
  (ind* on #1:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1632 for lem-bq3 and lem-bq4
    good-bit-queue(#1:BitQueue,B:Bit)
    = true
    if all-bits(#1:BitQueue ; B:Bit nil,B:Bit)= true .
)
---- assume lemmas lem-bq2 previously proved in abp.lemmata.itp
(goal ABP-PREDS+LEMMATA1632 : ABP-PREDS |-
  A{ #1:BitQueue ; B:Bit }
  (
    (A{ BQ:BitQueue ; B:Bit } ((all-bits(BQ ; (B nil),B)) = (all-bits(BQ,B)))) &
    (all-bits(#1:BitQueue ; B:Bit nil,B:Bit)) = (true)
    =>
    (good-bit-queue(#1:BitQueue,B:Bit)) = (true)
  )
.)
(cov on good-bit-queue(#1:BitQueue,B:Bit) .)
  (auto .)
  (ind on V0#0:Bit .)
    (ind* on V0#1:Bit .)
    (ind* on V0#1:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1633 for lem-bq3 and lem-bq5
    good-bit-queue(#1:BitQueue,B:Bit)
    = true
```

```
    if all-bits(#1:BitQueue ; B:Bit nil,flip(B:Bit))= true .
)
---- assume lemmas lem-bq2 previously proved in abp.lemmata.itp
(goal ABP-PREDS+LEMMATA1633 : ABP-PREDS |-
  A{ #1:BitQueue ; B:Bit }
  (
    (A{ BQ:BitQueue ; B:Bit } ((all-bits(BQ ; (B nil),B)) = (all-bits(BQ,B)))) &
    (all-bits(#1:BitQueue ; B:Bit nil,flip(B:Bit))) = (true)
    =>
    (good-bit-queue(#1:BitQueue,B:Bit)) = (true)
  )
.)
(lem aux : (A{ BQ:BitQueue } ((all-bits(BQ ; (on nil),off)) = (false)) ) .)
  (ind on BQ:BitQueue .)
    (auto .)
    (ind* on V0#0:Bit .)
(lem aux : (A{ BQ:BitQueue } ((all-bits(BQ ; (off nil),on)) = (false)) ) .)
  (ind on BQ:BitQueue .)
    (auto .)
    (ind* on V0#0:Bit .)
(cov on good-bit-queue(#1:BitQueue,B:Bit) .)
  (auto .)
  (ind on V0#0:Bit .)
    (ind* on V0#1:Bit .)
    (ind* on V0#1:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1636 for gbq-3 and lem-bq4
    all-bits(#2:BitQueue,B:Bit)
    = true
    if all-bits(B:Bit #2:BitQueue,B:Bit)= true .
)
(goal ABP-PREDS+LEMMATA1636 : ABP-PREDS |-
  A{ #2:BitQueue ; B:Bit }
  (
    (all-bits(B:Bit #2:BitQueue,B:Bit)) = (true)
    =>
    (all-bits(#2:BitQueue,B:Bit)) = (true)
  )
.)
(auto .)

---(
  ccp ABP-PREDS+LEMMATA1638 for gbq-3 and gbq-2
    all-bits(BQ:BitQueue,B1:Bit)
    = good-bit-queue(BQ:BitQueue,B1:Bit)
    if B1:Bit = flip(B1:Bit).
)
--- the equality enrichment predicate is used instead of equality;
--- the use of equality induces a loop in the rewriting process
(goal ABP-PREDS+LEMMATA1638 : ABP-PREDS |-
  A{ BQ:BitQueue ; B1:Bit }
  (
    (B1:Bit ~ flip(B1:Bit)) = (true)
    =>
```

```
    (good-bit-queue(BQ:BitQueue,B1:Bit)) = (all-bits(BQ:BitQueue,B1:Bit))
  )
.)
(ind* on B1:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1641 for lem-bq1 and lem-bq4
    good-bit-queue(#1:Bit #2:BitQueue,B:Bit)
    = true
    if all-bits(#1:Bit #1:Bit #2:BitQueue,B:Bit)= true .
)
(goal ABP-PREDS+LEMMATA1641 : ABP-PREDS |-
  A{ #2:BitQueue ; #1:Bit ; B:Bit }
  (
    (all-bits(#1:Bit #1:Bit #2:BitQueue,B:Bit)) = (true)
    =>
    (good-bit-queue(#1:Bit #2:BitQueue,B:Bit)) = (true)
  )
.)
(ind on B:Bit .)
  (ind* on #1:Bit .)
  (ind* on #1:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1648 for lem-pq3 and lem-pq4
    good-packet-queue(#1:BitPacketQueue,B:Bit,N:iNat)
    = true
    if all-packets(#1:BitPacketQueue ;(B:Bit,N:iNat)nil,B:Bit,N:iNat)= true .
)
(goal ABP-PREDS+LEMMATA1648 : ABP-PREDS |-
  A{ #1:BitPacketQueue ; B:Bit ; N:iNat }
  (
    (A{ BPQ:BitPacketQueue ; B:Bit ; N:iNat }
    ((all-packets(BPQ ; (B,N) nil,B,N)) = (all-packets(BPQ,B,N)))) &
    (all-packets(#1:BitPacketQueue ;(B:Bit,N:iNat)nil,B:Bit,N:iNat)) = (true)
    =>
    (good-packet-queue(#1:BitPacketQueue,B:Bit,N:iNat)) = (true)
  )
.)
---- assume lemma lem-pq2 previously proved in abp.lemmata.itp
(cov on good-packet-queue(#1:BitPacketQueue,B:Bit,N:iNat) .)
  (auto .)
  (ind on V0#0:Bit .)
    (ind* on V0#2:Bit .)
    (ind* on V0#2:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1649 for lem-pq3 and lem-pq5
    good-packet-queue(#1:BitPacketQueue,B:Bit,s N:iNat)
    = true
    if all-packets(#1:BitPacketQueue ;(B:Bit,s N:iNat)nil,flip(B:Bit),N:iNat)= true .
)
(goal ABP-PREDS+LEMMATA1649 : ABP-PREDS |-
  A{ #1:BitPacketQueue ; B:Bit ; N:iNat }
  (
```

```
    (A{ BPQ:BitPacketQueue ; B:Bit ; N:iNat }
    ((all-packets(BPQ ; (B,N) nil,B,N)) = (all-packets(BPQ,B,N)))) &
    (all-packets(#1:BitPacketQueue ;(B:Bit,s N:iNat)nil,flip(B:Bit),N:iNat)) = (true)
    =>
    (good-packet-queue(#1:BitPacketQueue,B:Bit,s N:iNat)) = (true)
  )
.)
(lem aux : (A{ BPQ:BitPacketQueue ; N:iNat ; N':iNat }
 ((all-packets(BPQ ; ((on,N') nil),off,N)) = (false)) ) .)
  (ind on BPQ:BitPacketQueue .)
    (auto .)
    (ind on V0#0:BitPacket .)
      (ind* on V1#0:Bit .)
(lem aux : (A{ BPQ:BitPacketQueue ; N:iNat ; N':iNat }
 ((all-packets(BPQ ; ((off,N') nil),on,N)) = (false)) ) .)
  (ind on BPQ:BitPacketQueue .)
    (auto .)
    (ind on V0#0:BitPacket .)
      (ind* on V1#0:Bit .)
(cov on good-packet-queue(#1:BitPacketQueue,B:Bit,s N:iNat) .)
  (auto .)
  (ind on V0#0:Bit .)
    (ind* on V0#2:Bit .)
    (ind* on V0#2:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1652 for lem-pq1 and lem-pq4
    good-packet-queue(#1:BitPacket #2:BitPacketQueue,B:Bit,N:iNat)
    = true
    if all-packets(#1:BitPacket #1:BitPacket #2:BitPacketQueue,B:Bit,N:iNat)= true .
)
(goal ABP-PREDS+LEMMATA1652 : ABP-PREDS |-
  A{ #1:BitPacket ; #2:BitPacketQueue ; B:Bit ; N:iNat }
  (
    (all-packets(#1:BitPacket #1:BitPacket #2:BitPacketQueue,B:Bit,N:iNat)) = (true)
    =>
    (good-packet-queue(#1:BitPacket #2:BitPacketQueue,B:Bit,N:iNat)) = (true)
  )
.)
(cov on good-packet-queue(#1:BitPacket #2:BitPacketQueue,B:Bit,N:iNat) .)
  (ind on V0#1:Bit .)
    (ind* on V0#3:Bit .)
    (ind* on V0#3:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1657 for gpq-3 and lem-pq4
    N:iNat ~ #2:iNat and all-packets(#3:BitPacketQueue,B:Bit,N:iNat)
    = true
    if all-packets((B:Bit,#2:iNat)#3:BitPacketQueue,B:Bit,N:iNat)= true .
)
(goal ABP-PREDS+LEMMATA1657 : ABP-PREDS |-
  A{ #2:iNat ; #3:BitPacketQueue ; B:Bit ; N:iNat }
  (
    (all-packets((B:Bit,#2:iNat)#3:BitPacketQueue,B:Bit,N:iNat)) = (true)
    =>
```

```
    (N:iNat ~ #2:iNat and all-packets(#3:BitPacketQueue,B:Bit,N:iNat)) = (true)
  )
.)
(auto .)

---(
  ccp ABP-PREDS+LEMMATA1659 for gpq-3 and gpq-2
    N:iNat ~ N':iNat and all-packets(BPQ:BitPacketQueue,B1:Bit,N:iNat)
    = N:iNat ~ s N':iNat and good-packet-queue(BPQ:BitPacketQueue,B1:Bit,N:iNat)
    if B1:Bit = flip(B1:Bit).
)
--- the equality enrichment predicate is used instead of equality;
--- the use of equality induces a loop in the rewriting process
(goal ABP-PREDS+LEMMATA1659 : ABP-PREDS |-
  A{ N':iNat ; BPQ:BitPacketQueue ; B1:Bit ; N:iNat }
  (
    (B1:Bit ~ flip(B1:Bit)) = (true)
    =>
    (N:iNat ~ N':iNat and all-packets(BPQ:BitPacketQueue,B1:Bit,N:iNat))
    = (N:iNat ~ s N':iNat and good-packet-queue(BPQ:BitPacketQueue,B1:Bit,N:iNat))
  )
.)
(ind* on B1:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1679 for lem-bq4 and gbq-2
    true
    = good-bit-queue(BQ:BitQueue,B:Bit)
    if B1:Bit = flip(B:Bit)/\ all-bits(B1:Bit BQ:BitQueue,B:Bit)= true .
)
--- the equality enrichment predicate is used instead of equality;
--- the use of equality induces a loop in the rewriting process
(goal ABP-PREDS+LEMMATA1679 : ABP-PREDS |-
  A{ BQ:BitQueue ; B:Bit ; B1:Bit }
  (
    (B1:Bit ~ flip(B:Bit)) = (true) &
    (all-bits(B1:Bit BQ:BitQueue,B:Bit)) = (true)
    =>
    (good-bit-queue(BQ:BitQueue,B:Bit)) = (true)
  )
.)
(ind on B:Bit .)
  (ind* on B1:Bit .)
  (ind* on B1:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1689 for gbq-2 and lem-bq4
    good-bit-queue(#2:BitQueue,B:Bit)
    = true
    if all-bits(#1:Bit #2:BitQueue,B:Bit)= true /\ #1:Bit = flip(B:Bit).
)
(goal ABP-PREDS+LEMMATA1689 : ABP-PREDS |-
  A{ #2:BitQueue ; B:Bit ; #1:Bit }
  (
    (#1:Bit ~ flip(B:Bit)) = (true) &
```

```
    (all-bits(#1:Bit #2:BitQueue,B:Bit)) = (true)
    =>
    (good-bit-queue(#2:BitQueue,B:Bit)) = (true)
  )
.)
(ind on B:Bit .)
  (ind* on #1:Bit .)
  (ind* on #1:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1698 for lem-pq4 and gpq-2
    true
    = N:iNat ~ s N':iNat and
      good-packet-queue(BPQ:BitPacketQueue,B:Bit,N:iNat)
    if B1:Bit = flip(B:Bit)

    /\ all-packets((B1:Bit,N':iNat)BPQ:BitPacketQueue,B:Bit,N:iNat)= true .
)
--- the equality enrichment predicate is used instead of equality;
--- the use of equality induces a loop in the rewriting process
(goal ABP-PREDS+LEMMATA1698 : ABP-PREDS |-
  A{ BPQ:BitPacketQueue ; B:Bit ; B1:Bit ; N:iNat ; N':iNat }
  (
    (B1:Bit ~ flip(B:Bit)) = (true) &
    (all-packets((B1:Bit,N':iNat)BPQ:BitPacketQueue,B:Bit,N:iNat)) = (true)
    =>
    (N:iNat ~ s N':iNat and
    good-packet-queue(BPQ:BitPacketQueue,B:Bit,N:iNat)) = (true)
  )
.)
(ind on B:Bit .)
  (ind* on B1:Bit .)
  (ind* on B1:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1705 for lem-pq5 and gpq-2
    true
    = N':iNat ~ #3:iNat and good-packet-queue(BPQ:BitPacketQueue,B:Bit,s #3:iNat)
    if B1:Bit = flip(B:Bit)
    /\ all-packets((B1:Bit,N':iNat)BPQ:BitPacketQueue,flip(B:Bit),#3:iNat)= true .
)
--- the equality enrichment predicate is used instead of equality;
--- the use of equality induces a loop in the rewriting process
---- assume lemma lem-pq5 previously proved in abp.lemmata.itp
(goal ABP-PREDS+LEMMATA1705 : ABP-PREDS |-
  A{ BPQ:BitPacketQueue ; B:Bit ; B1:Bit ; #3:iNat ; N':iNat }
  (
    (B1:Bit ~ flip(B:Bit)) = (true) &
    (all-packets((B1:Bit,N':iNat)BPQ:BitPacketQueue,flip(B:Bit),#3:iNat)) = (true) &
    (A{ BPQ:BitPacketQueue ; B:Bit ; N:iNat }
      ( (all-packets(BPQ,flip(B),N)) = (true) => (good-packet-queue(BPQ,B,s(N)))
        = (true) ))
    =>
    (N':iNat ~ #3:iNat and good-packet-queue(BPQ:BitPacketQueue,B:Bit,s #3:iNat))
        = (true)
```

```
  )
.)
(auto .)

---(
  ccp ABP-PREDS+LEMMATA1708 for gpq-2 and lem-pq4
    N:iNat ~ s #2:iNat and good-packet-queue(#3:BitPacketQueue,B:Bit,N:iNat)
    = true
    if all-packets((#1:Bit,#2:iNat)#3:BitPacketQueue,B:Bit,N:iNat)= true
    /\ #1:Bit = flip(B:Bit).
)
(goal ABP-PREDS+LEMMATA1708 : ABP-PREDS |-
  A{ #3:BitPacketQueue ; B:Bit ; #1:Bit ; #2:iNat ; N:iNat }
  (
    (#1:Bit ~ flip(B:Bit)) = (true) &
    (all-packets((#1:Bit,#2:iNat)#3:BitPacketQueue,B:Bit,N:iNat)) = (true)
    =>
    (N:iNat ~ s #2:iNat and
    good-packet-queue(#3:BitPacketQueue,B:Bit,N:iNat)) = (true)
  )
.)
(ind on B:Bit .)
  (ind* on #1:Bit .)
  (ind* on #1:Bit .)

---(
  ccp ABP-PREDS+LEMMATA1709 for gpq-2 and lem-pq5
    N:iNat ~ #2:iNat and good-packet-queue(#3:BitPacketQueue,B:Bit,s N:iNat)
    = true
    if all-packets((#1:Bit,#2:iNat)#3:BitPacketQueue,flip(B:Bit),N:iNat)= true
    /\ #1:Bit = flip(B:Bit).
)
(goal ABP-PREDS+LEMMATA1709 : ABP-PREDS |-
  A{ #3:BitPacketQueue ; B:Bit ; #1:Bit ; #2:iNat ; N:iNat }
  (
    (#1:Bit ~ flip(B:Bit)) = (true) &
    (all-packets((#1:Bit,#2:iNat)#3:BitPacketQueue,flip(B:Bit),N:iNat)) = (true) &
    (A{ BPQ:BitPacketQueue ; B:Bit ; N:iNat } ( (all-packets(BPQ,flip(B),N)) = (true)
      => (good-packet-queue(BPQ,B,s(N))) = (true) ))
    =>
    (N:iNat ~ #2:iNat and
     good-packet-queue(#3:BitPacketQueue,B:Bit,s N:iNat)) = (true)
  )
.)
(auto .)
```

Therefore `ABP-PREDS+LEMMATA` is admissible.

B.6 ITP Proof Scripts for Proof Obligations

This section contains the ITP proof scripts for completing the proof of `good-queues` $\wedge$ `inv` $\Rightarrow$ $\bigcirc$`inv`. This script and the output are available

in the `abp.itp` file.

```
---(
8. from inv-1a & recv-2b : pending
    gen-list(#5:iNat)~(#6:iNat #9:iNatList) = true
 if #5:iNat = #6:iNat
 /\ all-bits(#8:BitQueue,off) = true
 /\ all-packets(#7:BitPacketQueue,off,#5:iNat) = true
 /\ gen-list(#5:iNat) = #5:iNat #9:iNatList .
)
(goal po8 : ABP-PREDS |- A{ #5:iNat ; #6:iNat ; #9:iNatList ;
                            #8:BitQueue ; #7:BitPacketQueue }
 (
  (#5:iNat) = (#6:iNat) &
  (all-bits(#8:BitQueue,off)) = (true) &
  (all-packets(#7:BitPacketQueue,off,#5:iNat)) = (true) &
  (gen-list(#5:iNat)) = (#5:iNat #9:iNatList)
  =>
  (gen-list(#5:iNat) ~ (#6:iNat #9:iNatList)) = (true)
  )
.)
(auto .)

---(
46. from inv-1a & recv-2a : pending
    gen-list(#5:iNat)~(#6:iNat #9:iNatList) = true
 if #5:iNat = #6:iNat
 /\ all-bits(#8:BitQueue,on) = true
 /\ all-packets(#7:BitPacketQueue,on,#5:iNat) = true
 /\ gen-list(#5:iNat) = #5:iNat #9:iNatList .
)
(goal po46 : ABP-PREDS |- A{ #5:iNat ; #6:iNat ; #9:iNatList ;
                             #8:BitQueue ; #7:BitPacketQueue }
 (
  (#5:iNat) = (#6:iNat) &
  (all-bits(#8:BitQueue,on)) = (true) &
  (all-packets(#7:BitPacketQueue,on,#5:iNat)) = (true) &
  (gen-list(#5:iNat)) = (#5:iNat #9:iNatList)
  =>
  (gen-list(#5:iNat) ~ (#6:iNat #9:iNatList)) = (true)
  )
.)
(auto .)
```

B.7 ITP Proof Scripts for Lemmata

This section contains the ITP proof scripts for inductively proving in `ABP-PREDS` that the lemmata used in the ground stability proof of `good-queues` is indeed true. This script and the output are available in the `abp.lemmata.itp` file.

```
---(
  eq [lem-bq1] :
```

```
     good-bit-queue(B B BQ,B1)
   = good-bit-queue(B BQ,B1) .
)
(goal lem-bq1 : ABP-PREDS |- A{ B:Bit ; BQ:BitQueue ; B1:Bit }
  (
    (good-bit-queue(B B BQ,B1)) = (good-bit-queue(B BQ,B1))
  )
.)
(ind on B1:Bit .)
  (ind* on B:Bit .)
  (ind* on B:Bit .)

---(
  eq [lem-bq2] :
     all-bits(BQ ; (B nil),B)
   = all-bits(BQ,B) .
)
(goal lem-bq2 : ABP-PREDS |- A{ BQ:BitQueue ; B:Bit }
  (
    (all-bits(BQ ; (B nil),B)) = (all-bits(BQ,B))
  )
.)
(ind* on BQ:BitQueue .)

---(
  eq [lem-bq3] :
     good-bit-queue(BQ ; (B nil),B)
   = good-bit-queue(BQ,B) .
)
(goal lem-bq3 : ABP-PREDS |- A{ BQ:BitQueue ; B:Bit }
  (
    (good-bit-queue(BQ ; (B nil),B)) = (good-bit-queue(BQ,B))
  )
.)
(lem aux : (A{ BQ:BitQueue ; B:Bit }
 ((all-bits(BQ ; (B nil),B)) = (all-bits(BQ,B))) ) .)
  (ind* on BQ:BitQueue .)
(cov on good-bit-queue(BQ:BitQueue,B:Bit) .)
  (auto .)
  (ind on V0#0:Bit .)
    (ind* on V0#1:Bit .)
    (ind* on V0#1:Bit .)

---(
 ceq [lem-bq4] :
     good-bit-queue(BQ,B)
   = true
  if all-bits(BQ,B) = true .
)
(goal lem-bq4 : ABP-PREDS |- A{ BQ:BitQueue ; B:Bit }
  (
    (all-bits(BQ,B)) = (true)
    =>
    (good-bit-queue(BQ,B)) = (true)
  )
```

```
.)
(eq-split on good-bit-queue(BQ:BitQueue,B:Bit) .)
  (auto .)
  (ind on V0#0:Bit .)
    (ind* on V0#1:Bit .)
    (ind* on V0#1:Bit .)

---(
 ceq [lem-bq5] :
     good-bit-queue(BQ,B)
   = true
  if all-bits(BQ,flip(B)) = true .
)
(goal lem-bq5 : ABP-PREDS |- A{ BQ:BitQueue ; B:Bit }
  (
    (all-bits(BQ,flip(B))) = (true)
    =>
    (good-bit-queue(BQ,B)) = (true)
  )
.)
(cov on good-bit-queue(BQ:BitQueue,B:Bit) .)
  (auto .)
  (ind on V0#0:Bit .)
    (ind* on V0#1:Bit .)
    (ind* on V0#1:Bit .)

---(
  eq [lem-pq1] :
     good-packet-queue(BP BP BPQ,B,N)
   = good-packet-queue(BP BPQ,B,N) .
)
(goal lem-pq1 : ABP-PREDS |- A{ BPQ:BitPacketQueue ; B:Bit ;
                                BP:BitPacket ; N:iNat }
  (
    (good-packet-queue(BP BP BPQ,B,N)) = (good-packet-queue(BP BPQ,B,N))
  )
.)
(eq-split on good-packet-queue(BP:BitPacket BPQ:BitPacketQueue,B:Bit,N:iNat) .)
  (ind on V0#0:Bit .)
    (ind* on V0#3:Bit .)
    (ind* on V0#3:Bit .)

---(
  eq [lem-pq2] :
     all-packets(BPQ ; (B,N) nil,B,N)
   = all-packets(BPQ,B,N) .
)
(goal lem-pq2 : ABP-PREDS |- A{ BPQ:BitPacketQueue ; B:Bit ; N:iNat }
  (
    (all-packets(BPQ ; (B,N) nil,B,N)) = (all-packets(BPQ,B,N))
  )
.)
(cov* on all-packets(BPQ:BitPacketQueue,B:Bit,N:iNat) .)

---(
```

```
  eq [lem-pq3] :
     good-packet-queue(BPQ ; (B,N) nil,B,N)
   = good-packet-queue(BPQ,B,N) .
)
(goal lem-pq3 : ABP-PREDS |- A{ BPQ:BitPacketQueue ; B:Bit ; N:iNat }
  (
    (good-packet-queue(BPQ ; (B,N) nil,B,N)) = (good-packet-queue(BPQ,B,N))
  )
.)
(lem aux : (A{ BPQ:BitPacketQueue ; B:Bit ; N:iNat }
  ((all-packets(BPQ ; (B,N) nil,B,N)) = (all-packets(BPQ,B,N))) ) .)
  (cov* on all-packets(BPQ:BitPacketQueue,B:Bit,N:iNat) .)
(cov on good-packet-queue(BPQ:BitPacketQueue,B:Bit,N:iNat) .)
  (auto .)
  (ind on V0#0:Bit .)
    (ind* on V0#2:Bit .)
    (ind* on V0#2:Bit .)

---(
 ceq [lem-pq4] :
     good-packet-queue(BPQ,B,N)
   = true
  if all-packets(BPQ,B,N) .
)
(goal lem-pq4 : ABP-PREDS |- A{ BPQ:BitPacketQueue ; B:Bit ; N:iNat }
  (
    (all-packets(BPQ,B,N)) = (true)
    =>
    (good-packet-queue(BPQ,B,N)) = (true)
  )
.)
(cov on good-packet-queue(BPQ:BitPacketQueue,B:Bit,N:iNat) .)
  (auto .)
  (ind on V0#0:Bit .)
    (ind* on V0#2:Bit .)
    (ind* on V0#2:Bit .)

---(
 ceq [lem-pq5] :
     good-packet-queue(BPQ,B,s(N))
   = true
  if all-packets(BPQ,flip(B),N) = true .
)
(goal lem-pq5 : ABP-PREDS |- A{ BPQ:BitPacketQueue ; B:Bit ; N:iNat }
  (
    (all-packets(BPQ,flip(B),N)) = (true)
    =>
    (good-packet-queue(BPQ,B,s(N))) = (true)
  )
.)
(cov on good-packet-queue(BPQ:BitPacketQueue,B:Bit,s N:iNat) .)
  (auto .)
  (ind on V0#0:Bit .)
    (ind* on V0#2:Bit .)
    (ind* on V0#2:Bit .)
```

APPENDIX C

MISSING PROOFS FOR CHAPTER 6

This appendix documents the module structure of the IBOS specification and that of its predicates in Section 6; it also includes a proof of their admissibility.

The IBOS specification can be found in `ibos.maude` and the predicate specification in `ibos.preds.maude`, both available for download with this dissertation. The mechanical proofs were obtained with the tools integrated in the current version of the Maude Formal Environment (MFE) [36, 35].

C.1 Module Structure of `ibos.maude`

File `ibos.maude` comprises the Maude specification of the IBOS containing more than 1000 lines of code. This section describes the module structure of the specification, which is depicted in Figure C.1.

Modules `BOOL-OPS` and `CONFIGURATION` come from Maude's prelude. Natural numbers with their equality enrichment are defined in module `INAT`. Module `SYSCALL-TYPE` defines the constants modeling the different types of system calls available in the model. Labels, defined in module `LABEL` comprise constants such as `about-blank` and `url(iN:iNat)` for identifying web site labels. Module `MSG-TYPE` defines different types of messages such as `MSG-NEW-URL` or `MSG-RETURN-URL`, but not all of them are necessary for the formal verification. Module `PROC-ID` defines the infrastructure used for identifying processes either with a natural number `N:iNat` or a tuple `id(N:iNat)`. The top sort `Sys` of the specification is defined in module `SYS`. The payload associated to messages resulting from process interaction is modeled in module `PAYLOAD`. Module `MSG-PIPE-BASICS` defines the attributes for pipe objects, while modules `WEBAPPMGR` and `NETWORK` define

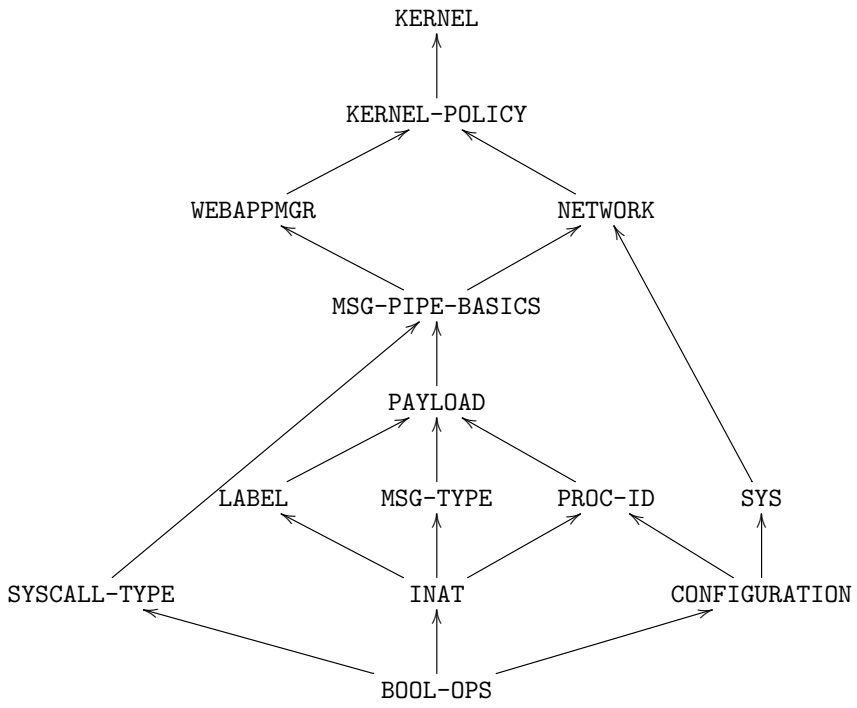

Figure C.1: Module inclusion in `ibos.maude`.

predicates for checking if a process is a web application or a network process, respectively. The object attributes for the kernel object are specified in module `KERNEL-POLICY`, together with auxiliary data types such as the policy multiset. All rewrite rules of the IBOS specification are contained in module `KERNEL`, which is the main module of the specification.

C.2 `IBOS` Admissibility and Free Constructors Modulo

This section presents the mechanical proofs for the admissibility of module `IBOS` and for the equational freeness of its subsignature of constructors.

A mechanical proof for ground operational termination could not be obtained automatically because of current limitations of the MTT tool. However, it is easy to see by inspection on the equations that the specification is ground operationally terminating. For ground sort-decreasingness, confluence, and coherence, the following is the output of the mechanical proof:

```
Maude> (ccr KERNEL .)
rewrites: 15472344 in 45954ms cpu (45950ms real) (336685 rewrites/second)
Church-Rosser check for KERNEL
  All critical pairs have been joined.
  The specification is locally-confluent.
  The module is sort-decreasing.
```

```
Maude> (cch KERNEL .)
rewrites: 2188028 in 1470ms cpu (1470ms real) (1487669 rewrites/second)
Coherence checking of KERNEL
  All critical pairs have been rewritten and no rewrite with rules can
  happen at non-overlapping positions of equations left-hand sides.
```

For sufficient completeness and equational freeness of constructors modulo, the following is the output of the mechanical proof:

```
Maude> (scc KERNEL .)
rewrites: 1027061 in 1048ms cpu (1046ms real) (979958 rewrites/second)
Sufficient completeness check for KERNEL
  Completeness counter-examples: none were found
  Freeness counter-examples: none were found
```

C.3 ibos.preds.maude

This section contains the Maude specification of the state predicates.

```
---- predicate defining initial states
mod IBOS-PRED-INIT is
  pr KERNEL .

  op init : Sys -> [Bool] .

  --- initial state
  eq [init]:
     init(
       { < 0 : nic | in(mtLL),out(mtLL) >
         < id(1) : kernel |
           msgPolicy(
             policy(id(3), id(4), MSG-FETCH-URL),
             policy(id(3), id(4), MSG-FETCH-URL-ABORT),
             policy(id(3), id(6), MSG-DOM-COOKIE-SET),
             policy(id(3), id(6), MSG-DOM-COOKIE-GET),
             policy(id(3), id(11), MSG-UI-MSG),
             policy(id(3), id(14), MSG-WRITE-FILE),
             policy(id(3), id(14), MSG-READ-FILE),
             policy(id(4), id(3), MSG-RETURN-URL),
             policy(id(4), id(3), MSG-RETURN-URL-METADATA),
             policy(id(4), id(6), MSG-COOKIE-SET),
             policy(id(4), id(6), MSG-COOKIE-GET),
             policy(id(6), id(3), MSG-DOM-COOKIE-GET-RETURN),
             policy(id(6), id(4), MSG-COOKIE-GET-RETURN),
             policy(id(11), id(3), MSG-NEW-URL),
             policy(id(11), id(3), MSG-SWITCH-TAB),
             policy(id(11), id(3), MSG-WEBAPP-MSG),
             policy(id(14), id(3), MSG-READ-FILE-RETURN),
             policy(id(14), id(11), MSG-DOWNLOAD-INFO)),
           nextNetworkProc(256),
           handledCurrently(none),
```

```
          weblabels(mtWPIS),
          networklabels(mtNPIS),
          displayedTopBar(about-blank) >
        < id(2) : proc | nextWAN(1024) >
        < id(2) : pipe | fromKernel(mt),toKernel(mt) >
        < id(5) : proc | none >
        < id(5) : pipe | fromKernel(mt),toKernel(mt) >
        < id(6) : proc | none >
        < id(6) : pipe | fromKernel(mt),toKernel(mt) >
        < id(7) : proc | none >
        < id(7) : pipe | fromKernel(mt), toKernel(mt) >
        < id(8) : proc | none >
        < id(8) : pipe | fromKernel(mt),toKernel(mt) >
        < id(9) : proc | none >
        < id(9) : pipe | fromKernel(mt),toKernel(mt) >
        < id(10) : proc | none >
        < id(10) : pipe | fromKernel(mt),toKernel(mt) >
        < id(11) : proc | none >
        < id(11) : pipe | fromKernel(mt),toKernel(mt) >
        < id(12) : proc | none >
        < id(12) : pipe | fromKernel(mt),toKernel(mt) >
        < id(13) : proc | none >
        < id(13) : pipe | fromKernel(mt),toKernel(mt) >
        < id(15) : proc | activeWebapp(id(0)),
                          displayedContent(about-blank) > })
   = true .
endm

---- unique kernel
mod IBOS-PRED-UNIQUE-KERNEL is
  pr KERNEL .

  var  Att         : AttributeSet .
  vars Att1 Att2   : AttributeSet .
  var  C           : Cid .
  var  Cnf         : Configuration .
  vars P P1 P2     : ProcId .

  op unique-kernel : Sys -> [Bool] .
 ceq [unique-kernel-0] :
     unique-kernel( { Cnf } )
   = false
  if no-kernel(Cnf) .
  eq [unique-kernel-1] :
     unique-kernel( { < P : kernel | Att > Cnf } )
   = no-kernel(Cnf) .
  eq [unique-kernel-2] :
     unique-kernel( { < P1 : kernel | Att1 >
                      < P2 : kernel | Att2 > Cnf } )
   = false .

  --- auxiliary function symbol
  op no-kernel : Configuration -> Bool .
```

```
  eq no-kernel(none)
   = true .
  eq no-kernel(<> Cnf)
   = no-kernel(Cnf) .
  eq no-kernel( < P : C | Att > Cnf )
   = not(C ~ kernel) and no-kernel(Cnf) .
endm

---- policy immutability
mod IBOS-PRED-IMMUTABLE-POLICY is
  pr KERNEL .

  op immutable-policy : Sys PolicySet -> [Bool] .

  op init-policy : -> PolicySet .
  eq init-policy
   = ( policy(id(3), id(4), MSG-FETCH-URL),
       policy(id(3), id(4), MSG-FETCH-URL-ABORT),
       policy(id(3), id(6), MSG-DOM-COOKIE-SET),
       policy(id(3), id(6), MSG-DOM-COOKIE-GET),
       policy(id(3), id(11), MSG-UI-MSG),
       policy(id(3), id(14), MSG-WRITE-FILE),
       policy(id(3), id(14), MSG-READ-FILE),
       policy(id(4), id(3), MSG-RETURN-URL),
       policy(id(4), id(3), MSG-RETURN-URL-METADATA),
       policy(id(4), id(6), MSG-COOKIE-SET),
       policy(id(4), id(6), MSG-COOKIE-GET),
       policy(id(6), id(3), MSG-DOM-COOKIE-GET-RETURN),
       policy(id(6), id(4), MSG-COOKIE-GET-RETURN),
       policy(id(11), id(3), MSG-NEW-URL),
       policy(id(11), id(3), MSG-SWITCH-TAB),
       policy(id(11), id(3), MSG-WEBAPP-MSG),
       policy(id(14), id(3), MSG-READ-FILE-RETURN),
       policy(id(14), id(11), MSG-DOWNLOAD-INFO)) .

  var  Att    : AttributeSet .
  var  Cnf    : Configuration .
  var  PS     : PolicySet .

  eq [immutable-policy]:
     immutable-policy({ < id(1) : kernel | msgPolicy(PS), Att >
                        Cnf }, PS)
   = unique-kernel({ < id(1) : kernel | msgPolicy(PS), Att >
                     Cnf } ) .
endm

---- all predicates
mod IBOS-PREDS is
  pr IBOS-PRED-INIT .
  pr IBOS-PRED-UNIQUE-KERNEL .
  pr IBOS-PRED-IMMUTABLE-POLICY .
endm
```

C.4 IBOS-PREDS is Admissible

This section presents the mechanical proofs for the admissibility of module IBOS-PREDS.

A mechanical proof for ground operational termination could not be obtained automatically because of current limitations of the MTT tool. However, it is easy to see by inspection on the equations that the specification is ground operationally terminating. For ground sort-decreasingness, confluence, and coherence, the following is the output of the mechanical proof:

```
Maude> (ccr IBOS-PREDS .)
rewrites: 17325456 in 198165253ms cpu (198213314ms real) (87 rewrites/second)
Church-Rosser check for IBOS-PREDS
  All critical pairs have been joined.
  The specification is locally-confluent.
  The module is sort-decreasing.

Maude> (cch IBOS-PREDS .)
rewrites: 18767431 in 220183279ms cpu (230455430ms real) (85 rewrites/second)
Coherence checking of IBOS-PREDS
  All critical pairs have been rewritten and no rewrite with rules can happen
  at non-overlapping positions of equations left-hand sides.
```

REFERENCES

[1] M. AlTurki and J. Meseguer. Reduction semantics and formal analysis of Orc programs. *Electronic Notes in Theoretical Computer Science*, 200(3):25–41, 2008.

[2] J. Avenhaus, T. Hillenbrand, and B. Löchner. On using ground joinable equations in equational theorem proving. *Journal of Symbolic Computation*, 36(1-2):217–233, 2003.

[3] M. Ayala-Rincón. *Expressiveness of Conditional Equational Systems with Built-in Predicates*. PhD thesis, Universität Kaiserslauten, 1993.

[4] F. Baader, editor. *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*. Springer, 2007.

[5] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[6] L. Bachmair, N. Dershowitz, and D. A. Plaisted. Completion without failure. In A. H. Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, pages 1–30. Academic Press, New York, 1989.

[7] D. Balasubramanian, C. Păsăreanu, M. W. Whalen, G. Karsai, and M. R. Lowry. Polyglot: modeling and analysis for multiple Statechart formalisms. In M. B. Dwyer and F. Tip, editors, *ISSTA*, pages 45–55. ACM, 2011.

[8] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commununications of the ACM*, 12(5):260–261, May 1969.

[9] K. Becker. Proving ground confluence and inductive validity in constructor based equational specifications. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT*, volume 668 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 1993.

[10] J. Bergstra and J. Klop. Verification of an alternating bit protocol by means of process algebra protocol. In W. Bibel and K. Jantke, editors, *Mathematical Methods of Specification and Synthesis of Software Systems '85*, volume 215 of *Lecture Notes in Computer Science*, pages 9–23. Springer Berlin / Heidelberg, 1986.

[11] G. Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press.

[12] M. Bezem and J. F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *CONCUR*, volume 836 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 1994.

[13] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu. A systematic approach to model checking human-automation interaction using task analytic models. *IEEE Transactions on Systems, Man, and Cybernetics–Part A: Systems and Humans*, 41(5):961–976, 2011.

[14] A. Bouhoula. Using induction and rewriting to verify and complete parameterized specifications. *Theoretical Computer Science*, 170(1-2):245–276, 1996.

[15] A. Bouhoula. Simultaneous checking of completeness and ground confluence for algebraic specifications. *ACM Transactions on Computational Logic*, 10(3), 2009.

[16] A. Bouhoula and F. Jacquemard. Automated induction with constrained tree automata. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 539–554. Springer, 2008.

[17] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1-2):35–132, 2000.

[18] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[19] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188. ACM, 1987.

[20] K. M. Chandy and J. Misra. *Parallel Program Design, A foundation.* Addison Wesley 1988, 1988.

[21] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang. A systematic approach to uncover security flaws in GUI logic. In *IEEE Symposium on Security and Privacy*, pages 71–85. IEEE Computer Society, 2007.

[22] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[23] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[24] M. Clavel and M. Egea. Itp/ocl: A rewriting-based validation tool for uml+ocl static class diagrams. In Johnson and Vene [57], pages 368–373.

[25] H. Comon. Sufficient completness, term rewriting systems and "anti-unification". In J. H. Siekmann, editor, *CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 1986.

[26] H. Comon. An effective method for handling initial algebras. In J. Grabowski, P. Lescanne, and W. Wechler, editors, *ALP*, volume 343 of *Lecture Notes in Computer Science*, pages 108–118. Springer, 1988.

[27] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007.

[28] H. Comon and F. Jacquemard. Ground reducibility is EXPTIME-complete. *Information and Computation*, 187(1):123–153, 2003.

[29] G. Dowek, C. Muñoz, and C. Păsăreanu. A formal analysis framework for PLEXIL. In *Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems*, pages 45–51, September 2007.

[30] G. Dowek, C. Muñoz, and C. Păsăreanu. A small-step semantics of PLEXIL. Technical Report 2008-11, National Institute of Aerospace, Hampton, VA, 2008.

[31] G. Dowek, C. Muñoz, and C. Rocha. Rewriting logic semantics of a plan execution language. *CoRR*, abs/1002.2872, 2010.

[32] F. Durán, S. Lucas, and J. Meseguer. Termination modulo combinations of equational theories. In S. Ghilardi and R. Sebastiani, editors, *FroCos*, volume 5749 of *Lecture Notes in Computer Science*, pages 246–262. Springer, 2009.

[33] F. Durán and J. Meseguer. A Church-Rosser checker tool for conditional order-sorted equational maude specifications. In P. C. Ölveczky, editor, *WRLA*, volume 6381 of *Lecture Notes in Computer Science*, pages 69–85. Springer, 2010.

[34] F. Durán and J. Meseguer. On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *Journal of Logic and Algebraic Programming*, to appear, 2011.

[35] F. Durán, C. Rocha, and J. M. Álvarez. Tool interoperability in the maude formal environment. In *Algebra and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*, pages 400–406, 2011.

[36] F. Durán, C. Rocha, and J. M. Álvarez. Towards a maude formal environment. In *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 329–351, 2011.

[37] S. Escobar, C. Meadows, and J. Meseguer. State space reduction in the maude-nrl protocol analyzer. *CoRR*, abs/1105.5282, 2011.

[38] S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In Baader [4], pages 153–168.

[39] T. Estlin, A. Jónsson, C. Păsăreanu, R. Simmons, K. Tso, and V. Verma. Plan Execution Interchange Language (PLEXIL). Technical Memorandum TM-2006-213483, NASA, 2006.

[40] S. Falke and D. Kapur. Operational termination of conditional rewriting with built-in numbers and semantic data structures. *Electronic Notes in Theoretical Computer Science*, 237:75–90, 2009.

[41] S. Falke and D. Kapur. Rewriting induction + linear arithmetic = decision procedure. In B. Gramlich, D. Miller, and U. Sattler, editors, *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 241–255. Springer, 2012.

[42] A. Farzan and J. Meseguer. State space reduction of rewrite theories using invisible transitions. In Johnson and Vene [57], pages 142–157.

[43] E. Giménez. An application of co-inductive types in coq: Verification of the alternating bit protocol. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*, pages 135–152. Springer Berlin / Heidelberg, 1996.

[44] I. Gnaedig and H. Kirchner. Computing constructor forms with non terminating rewrite programs. In A. Bossi and M. J. Maher, editors, *PPDP*, pages 121–132. ACM, 2006.

[45] I. Gnaedig and H. Kirchner. Computing Constructor Forms with Non Terminating Rewrite Programs -Extended version-. Research Report, PROTHEO - INRIA Lorraine - LORIA - INRIA - CNRS : UMR7503 - Université Henri Poincaré - Nancy I - Université Nancy II - Institut National Polytechnique de Lorraine, 2006.

[46] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[47] C. Grier, S. Tang, and S. T. King. Designing and implementing the OP and OP2 web browsers. *TWEB*, 5(2):11, 2011.

[48] R. Gutiérrez, J. Meseguer, and C. Rocha. Order-sorted equality enrichments modulo axioms (extended version). Technical report, University of Illinois at Urbana-Champaing, December 2011. Available at `http://hdl.handle.net/2142/28597`.

[49] R. Gutiérrez, J. Meseguer, and C. Rocha. Order-sorted equality enrichments modulo axioms. In F. Durán, editor, *Rewriting Logic and Its Applications*, volume 7571 of *Lecture Notes in Computer Science*, pages 162–181. Springer Berlin Heidelberg, 2012.

[50] J. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, University of Toronto, Computer Science Department, 1975.

[51] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[52] J. Hendrix. *Decision Procedures for Equationally Based Reasoning*. PhD thesis, University of Illinois at Urbana-Champaign, April 2008.

[53] J. Hendrix, M. Clavel, and J. Meseguer. A sufficient completeness reasoning tool for partial specifications. In J. Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 165–174. Springer, 2005.

[54] J. Hendrix and J. Meseguer. On the completeness of context-sensitive order-sorted specifications. In Baader [4], pages 229–245.

[55] J. Hendrix, H. Ohsaki, and M. Viswanathan. Propositional tree automata. In F. Pfenning, editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2006.

[56] G. P. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. In *FOCS*, pages 96–107. IEEE, 1980.

[57] M. Johnson and V. Vene, editors. *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings*, volume 4019 of *Lecture Notes in Computer Science*. Springer, 2006.

[58] J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In J. Díaz, editor, *ICALP*, volume 154 of *Lecture Notes in Computer Science*, pages 361–373. Springer, 1983.

[59] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82(1):1–33, 1989.

[60] D. Kapur, P. Narendran, and F. Otto. On ground-confluence of term rewriting systems. *Information and Computation*, 86(1):14–31, 1990.

[61] D. Kapur, P. Narendran, D. J. Rosenkrantz, and H. Zhang. Sufficient-completeness, ground-reducibility and their complexity. *Acta Informatica*, 28(4):311–350, 1991.

[62] D. Kapur, P. Narendran, and H. Zhang. On sufficient-completeness and related properties of term rewriting systems. *Acta Informatica*, 24(4):395–415, 1987.

[63] E. Kounalis. Testing for the ground (co-)reducibility property in term-rewriting systems. *Theoretical Computer Science*, 106(1):87–117, 1992.

[64] A. Lazrek, P. Lescanne, and J.-J. Thiel. Tools for proving inductive equalities, relative completeness, and omega-completeness. *Information and Computation*, 84(1):47–70, 1990.

[65] K. Lin and J. Goguen. A hidden proof of the alternating bit protocol. Available at `http://cseweb.ucsd.edu/~goguen/pps/abp.ps`.

[66] D. Lucanu. Strategy-based rewrite semantics for membrane systems preserves maximal concurrency of evolution rule actions. *Electronic Notes in Theoretical Computer Science*, 237:107–125, 2009.

[67] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[68] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems*. Springer-Verlag, New York, 1995.

[69] U. Martin and T. Nipkow. Ordered rewriting and confluence. In M. E. Stickel, editor, *CADE*, volume 449 of *Lecture Notes in Computer Science*, pages 366–380. Springer, 1990.

[70] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[71] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *WADT*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.

[72] J. Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, (to appear), 2012.

[73] J. Meseguer and J. A. Goguen. Initially, induction and computability. *Algebraic Methods in Semantics*, 1986.

[74] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theoretical Computer Science*, 403(2-3):239–264, 2008.

[75] J. Meseguer and G. Rosu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[76] J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Monographs in Computer Science. Springer-Verlag, New York, 2001.

[77] T. Nipkow. Combining matching algorithms: The regular case. *Journal of Symbolic Computation*, 12(6):633–654, 1991.

[78] T. Nipkow and G. Weikum. A decidability result about sufficient-completeness of axiomatically specified abstract data types. In A. B. Cremers and H.-P. Kriegel, editors, *Theoretical Computer Science*, volume 145 of *Lecture Notes in Computer Science*, pages 257–268. Springer, 1983.

[79] I. of Computer Science University of Innsbruck. Constrained rewriting and smt: Emerging trends in rewriting. Available at `http://cl-informatik.uibk.ac.at/research/projects/constrained-rewriting-and-smt-emerging-trends-in`.

[80] K. Ogata and K. Futatsugi. Proof scores in the OTS/CafeOBJ Method. In E. Najm, U. Nestmann, and P. Stevens, editors, *FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 2003.

[81] K. Ogata and K. Futatsugi. Simulation-based verification for invariant properties in the ots/cafeobj method. *Electronic Notes in Theorethical Computer Science*, 201:127–154, 2008.

[82] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[83] D. Plaisted. Semantic confluence tests and completion methods. *Information and Control*, 65:182–215, 1985.

[84] C. Rocha, H. Cadavid, C. A. Muñoz, and R. Siminiceanu. A formal interactive verification environment for the plan execution interchange language. In J. Derrick, S. Gnesi, D. Latella, and H. Treharne, editors, *IFM*, volume 7321 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2012.

[85] C. Rocha and J. Meseguer. Theorem proving modulo based on boolean equational procedures. In R. Berghammer, B. Möller, and G. Struth, editors, *RelMiCS*, volume 4988 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2008.

[86] C. Rocha and J. Meseguer. Constructors, sufficient completeness and deadlock freedom of generalized rewrite theories. Technical report, University of Illinois at Urbana-Champaign, 2010. Available at `http://hdl.handle.net/2142/15474`.

[87] C. Rocha and J. Meseguer. Constructors, sufficient completeness, and deadlock freedom of rewrite theories. In C. G. Fermüller and A. Voronkov, editors, *LPAR (Yogyakarta)*, volume 6397 of *Lecture Notes in Computer Science*, pages 594–609. Springer, 2010.

[88] C. Rocha and J. Meseguer. Proving safety properties of rewrite theories. Technical report, University of Illinois at Urbana-Champaign, 2010. Available at `http://hdl.handle.net/2142/17407`.

[89] C. Rocha and J. Meseguer. Proving safety properties of rewrite theories. In A. Corradini, B. Klin, and C. Cîrstea, editors, *CALCO*, volume 6859 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2011.

[90] C. Rocha, C. Munoz, and H. Cadavid. A graphical environment for the semantic validation of a plan execution language. In *Space Mission Challenges for Information Technology, 2009. SMC-IT 2009. Third IEEE International Conference on*, pages 201 –207, july 2009.

[91] C. Rocha, C. Muñoz, and G. Dowek. A formal library of set relations and its application to synchronous languages. *Theoretical Computer Science*, 412(37):4853 – 4866, 2011.

[92] C. Rocha and C. A. Muñoz. Simulation and verification of synchronous set relations in rewriting logic. In A. da Silva Simão and C. Morgan, editors, *SBMF*, volume 7021 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2011.

[93] G. Roşu and A. Ştefănescu. Matching Logic: A New Program Verification Approach (NIER Track). In *ICSE'11: Proceedings of the 30th International Conference on Software Engineering*, pages 868–871. ACM, 2011.

[94] V. Rusu. Combining theorem proving and narrowing for rewriting-logic specifications. In G. Fraser and A. Gargantini, editors, *TAP*, volume 6143 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2010.

[95] R. Sasse. *Security Models in Rewriting Logic for Cryptographic Protocols and Browsers*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.

[96] R. Sasse, S. T. King, J. Meseguer, and S. Tang. IBOS: A correct-by-construction modular browser. In C. Păsăreanu and G. Salaün, editors, *FACS*, volume 7684 of *Lecture Notes in Computer Science*, pages 224–241. Springer, 2012.

[97] T. Serbanuta, G. Stefanescu, and G. Rosu. Defining and executing P systems with structured data in K. In D. W. Corne, P. Frisco, G. Paun, G. Rozenberg, and A. Salomaa, editors, *Workshop on Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 374–393. Springer, 2008.

[98] L. Steggles and P. Kosiuczenko. A timed rewriting logic semantics for sdl: A case study of the alternating bit protocol. *Electronic Notes in Theoretical Computer Science*, 15(0):83 – 104, 1998.

[99] P. J. Strauss. Executable semantics for PLEXIL: simulating a task-scheduling language in Haskell. Master's thesis, Oregon State University, 2009.

[100] I. Suzuki. Formal analysis of the alternating bit protocol by temporal petri nets. *IEEE Transactions on Software Engineering*, 16(11):1273–1281, 1990.

[101] S. Tang, H. Mai, and S. T. King. Trust and protection in the illinois browser operating system. In R. H. Arpaci-Dusseau and B. Chen, editors, *OSDI*, pages 17–32. USENIX Association, 2010.

[102] O. Tardieu. A deterministic logical semantics for pure esterel. *ACM Transactions On Programming Languages and Systems*, 29(2):8, 2007.

[103] V. Verma, A. Jónsson, C. Păsăreanu, and M. Iatauro. Universal Executive and PLEXIL: Engine and language for robust spacecraft control and operations. In *Proceedings of the American Institute of Aeronautics and Astronautics Space Conference*, 2006.