Santiago Escobar (Ed.)

Rewriting Logic and its Applications

10th International Workshop, WRLA 2014

part of the European Joint Conferences on Theory & Practice of Software (ETAPS 2014)

Grenoble, France, April 5^{th} and 6^{th}, 2014.

Informal Proceedings

Preface

This volume contains the informal proceedings of the *10th International Workshop on Rewriting Logic and its Applications* (WRLA 2014),held on April 5th and 6th 2014 in Grenoble, France.

Rewriting logic (RL) is a natural model of computation and an expressive semantic framework for concurrency, parallelism, communication, and interaction. It can be used for specifying a wide range of systems and languages in various application fields. It also has good properties as a metalogical framework for representing logics. In recent years, several languages based on RL (ASF+SDF, CafeOBJ, ELAN, Maude) have been designed and implemented. The aim of the workshop is to bring together researchers with a common interest in RL and its applications, and to give them the opportunity to present their recent works, discuss future research directions, and exchange ideas. The previous meetings were held on Asilomar (USA) 1996, Pont-a-Mousson (France) 1998, Kanazawa (Japan) 2000, Pisa (Italy) 2002, Barcelona (Spain) 2004, Vienna (Austria) 2006, Budapest (Hungary) 2008, Paphos (Cyprus) 2010, and Tallinn (Estonia) 2012.

Typically, the topics of interest include (but are not restricted to):

- foundations and models of RL;
- languages based on RL, including implementation issues;
- RL as a logical framework;
- RL as a semantic framework, including applications of RL to
  - object-oriented systems,
  - concurrent and/or parallel systems,
  - interactive, distributed, open ended and mobile systems,
  - specification of languages and systems;
- use of RL to provide rigorous support for model-based software engineering;
- formalisms related to RL, including
  - real-time and probabilistic extensions of RL,
  - rewriting approaches to behavioral specifications,
  - tile logic;
- verification techniques for RL specifications, including
  - equational and coherence methods,
  - verification of properties expressed in first-order, higher-order, modal and temporal logics,
  - narrowing-based analysis and verification;
- comparisons of RL with existing formalisms having analogous aims;
- application of RL to specification and analysis of
  - distributed systems,
  - physical systems.

The last editions of WRLA were held as a satellite event of the European Joint Conferences on Theory & Practice of Software (ETAPS). This year edition is a satellite event of ETAPS 2014.

The revised versions of the contributions selected to be presented at the workshop are included in this informal proceedings. Each contribution was reviewed by at least three Program Committee members. This volume also includes the abstracts of four invited speakers: Francisco Durán from the Universidad de Málaga, Spain, Alberto Lluch-Lafuente from the IMT Institute for Advanced Studies Lucca, Italy, Peter Ölveczky from the University of Oslo, Norway, and Cesare Tinelli from the University of Iowa, USA. We would like to thank them for having accepted our invitation.

We would also like to thank all the members of the Program Committee and all the referees for their careful work in the review process. Finally, I express our gratitude to all members of the local organization of ETAPS 2014 and the Easychair system, whose work has made the workshop possible.

March 2014 Santiago Escobar

Table of Contents

Program Committee

Roberto Bruni	Dipartimento di Informatica, Università di Pisa, Italy
Manuel Clavel	Universidad Complutense de Madrid, Spain
Francisco Durán	University of Málaga, Spain
Santiago Escobar	Universitat Politècnica de València, Spain
Kokichi Futatsugi	JAIST, Japan
Alexander Knapp	Universität Augsburg, Germany
Alberto Lluch Lafuente	IMT Institute for Advanced Studies Lucca, Italy
Dorel Lucanu	Alexandru Ioan Cuza University, Rumania
Narciso Martí Oliet	Universidad Complutense de Madrid, Spain
Jose Meseguer	University of Illinois at Urbana-Champaign, USA
Ugo Montanari	Dipartimento di Informatica - Università di Pisa, Italy
Pierre-Etienne Moreau	Ecole des Mines de Nancy INRIA Nancy, France
Kazuhiro Ogata	JAIST, Japan
Peter Olveczky	University of Oslo, Norway
Miguel Palomino	Universidad Complutense de Madrid, Spain
Grigore Rosu	University of Illinois at Urbana-Champaign, USA
Vlad Rusu	INRIA Lille Nord-Europe, France
Mark-Oliver Stehr	SRI International, USA
Carolyn Talcott	SRI International, USA
Mark van Den Brand	Eindhoven University of Technology, The Netherlands
Martin Wirsing	Ludwig-Maximilians-Universitaet Muenchen, Germany

Additional Reviewers

A

Abd Alrahman, Yehia
Aguirre, Luis
Asavoae, Irina Mariuca
B

Bodei, Chiara
Bosnacki, Dragan
C

Calvès, Christophe François Olivier
G

Gadducci, Fabio
M

Martin, Oscar
R

Riesco, Adrián
S

Sakai, Masahiko
Sammartino, Matteo
Stefanescu, Andrei
V

Vandin, Andrea
W

Wijs, Anton

Author Index

On the composition of graph-transformation-based DSL definitions

Francisco Durán

Department of Computer Science, University of Málaga
duran@lcc.uma.es

Abstract. Model-driven engineering (MDE) is an increasingly popular approach to systems development. However, its two main ingredients, namely domain-specific languages (DSLs) and model transformations currently lack of the appropriate formal background as to allow their complete development. DSLs are particularly interesting because they allow encoding domain-knowledge into a modelling language and enable full code generation and analysis based on high-level models. Moreover, as a result of the domain-specificity of DSLs, there is a need for many such languages, which means that their use only becomes economically viable if the development of new DSLs can be made efficient. We present results on the modularity of DSLs whose behaviour is specified through in-place model transformations. Specifically, we present a formal framework of morphisms between graph-transformation systems (GTSs) that allow us to define a novel technique for conservative extensions of such DSLs. We illustrate the use of some of these results in the context of the modular specification of non-functional properties of systems. We apply our approach for the specification and monitoring of non-functional properties using generic observers DSLs, which can be used to analyse the required non-functional properties of the system with the guarantee of not changing its behaviour. The approach has been used for the definition of a modular, model-based partial reimplementation of one well-known analysis framework, namely the Palladio Architecture Simulator.

Can we efficiently check concurrent programs under relaxed memory models in Maude?

Alberto LLuch-Lafuente

IMT Institute for Advanced Studies, Italy
alberto.lluch@imtlucca.it

Abstract. Relaxed memory models offer suitable abstractions of the actual optimizations offered by multi-core architectures and by compilers of concurrent programming languages. Using such abstractions for verification purposes is challenging in part since they introduce yet another source of high non-determinism, thus contributing to the state explosion problem. In the last years several techniques have been proposed to mitigate those problems so to make verification under relaxed memory models feasible. I would like to present some of those techniques and to discuss if and how those techniques can be adopted in Maude or Maude-based verification tools.

Real-Time Maude and its Applications

Peter Ölveczky

Department of Informatics, University of Oslo, Norway
peterol@ifi.uio.no

Abstract. Real-Time Maude extends the rewriting-logic-based Maude system to support the executable formal modeling and analysis of real-time systems. Real-Time Maude is characterized by its general and expressive, yet intuitive, specification formalism, and offers a spectrum of formal analysis methods, including: rewriting for simulation purposes, search for reachability analysis, and both untimed and metric temporal logic model checking. Real-Time Maude is particularly suitable for specifying real-time systems in an object-oriented style, and its flexible formalism makes it easy to model different forms of communication.
This modeling flexibility, and the usefulness of Real-Time Maude for both simulation and model checking, has been demonstrated on advanced state-of-the-art applications, including both distributed protocols of different kinds as well as industrial embedded systems. Furthermore, Real-Time Maude's expressiveness has also been exploited for defining the formal semantics of a number of modeling languages for real-time/embedded systems. Real-Time Maude thereby provides formal model checking capabilities for these languages.
This tutorial gives an overview of Real-Time Maude and some of its applications, and mentions some future research challenges.

Extending SMT solving with constrained deduction and rewrite rules

Cesare Tinelli

Department of Computer Science, The University of Iowa
cesare-tinelli @ uiowa.edu

Abstract. SMT solvers are very effective at reasoning about ground formulas over a variety of theories such as linear arithmetic, the theory of equality, of arrays, of bit bit vectors, and so on. Their high performance with respect to other kinds of automated provers is achieved by building in specialized reasoning techniques for these theories. To reason about non-built-in theories users of SMT solvers must provide as input quantified formulas axiomatizing those theories, as with general-purpose theorem provers. Unfortunately, combining built-in theory reasoning with quantifier reasoning is extremely challenging, both in principle and in practice. As a result, SMT solvers accepting quantified formulas rely on heuristic, and generally incomplete, techniques for generating selected ground instances of those formulas. As an alternative to this approach, we present an abstract framework for extending SMT solvers with user-specified guarded rewrite rules and deduction rules for reasoning about non-built-in symbols. These rules, which are guarded by constraints over the built-in theories, are similar in spirit to Constraint Handling Rules in Logic Programming and to the recently introduced Logically Constrained Term Rewriting Systems. In concrete, the framework allows one to fully integrate a powerful rewrite/deduction engine inside the general DPLL(T) architecture used by most SMT solvers. We conjecture that in several cases, especially for problems coming for program verification, using this approach to extend built-in theories is more effective in practice than relying on quantified axioms and heuristic quantifier instantiation. We will present initial experimental evidence supporting this claim obtained from a first implementation of our framework within the CVC4 SMT solver.
This is joint work with Andrew Reynolds and Franois Bobot.

Conditional Narrowing Modulo in Rewriting Logic and Maude*

Luis Aguirre, Narciso Martí-Oliet, Miguel Palomino, and Isabel Pita

Facultad de Informática, Universidad Complutense de Madrid, Spain
{luisagui, narciso, miguelpt, ipandreu}@ucm.es

Abstract. This work studies the relationship between verifiable and computable answers for reachability problems in rewrite theories with an underlying membership equational logic. These problems have the form $(\exists \bar{x}) s(\bar{x}) \rightarrow^* t(\bar{x})$, with $\bar{x}$ some variables, or a conjunction of several of these subgoals. A calculus that solves this kind of problems has been developed and proved correct. Given a reachability problem in a rewrite theory, this calculus can compute any normalized answer that can be checked by rewriting, or a more general one. Special care has been taken in the calculus to keep membership information attached to each term, using this information whenever possible.

Keywords: Maude, narrowing, reachability, rewriting logic, unification, membership equational logic

1 Introduction

Rewriting logic is a computational logic that has been around for more than twenty years [Mes90], whose semantics [BM06] has a precise mathematical meaning allowing mathematical reasoning for property proving, providing a more flexible framework for the specification of concurrent systems. It turned out that it can express both concurrent computation and logical deduction, allowing its application in many areas such as automated deduction, software and hardware specification and verification, security, etc. One important property of rewriting logic is reflection [CM96]. Intuitively, reflection means representing a logic's metalevel at the object level, allowing the definition of strategies that guide rule application in an object-level theory.

Reachability problems have the form $(\exists \bar{x}) s(\bar{x}) \rightarrow^* t(\bar{x})$, with $\bar{x}$ some variables, or a conjunction of several of these subgoals. They can be solved by model checking methods for finite state spaces. A technique known as *narrowing* [Fay78] that was first proposed as a method for solving equational goals (*unification*), has been extended to cover also reachability goals [MT07], leaving equational goals as a special case of reachability goals. In recent years the idea of *variants of a term* has been applied to narrowing. A strategy for order-sorted unconditional

* Research supported by MINECO Spanish project StrongSoft (TIN2012–39391–C04–04) and Comunidad de Madrid program PROMETIDOS (S2009/TIC-1465).

rewrite theories known as *folding variant narrowing* [ESM12], which computes a complete set of variants of any term, has been developed by Escobar, Sasse and Meseguer, allowing unification *modulo* a set of equations and axioms. The strategy terminates on any input term on those systems enjoying the *finite variant property*, and it is *optimally terminating*. It is being used for cryptographic protocol analysis [MT07], with tools like Maude-NPA [EMM05], termination algorithms modulo axioms [DLM+08], and algorithms for checking confluence and coherence of rewrite theories modulo axioms, such as the Church-Rosser (CRC) and the Coherence (ChC) Checkers for Maude [DM12].

This work explores narrowing for membership conditional rewrite theories, going beyond the scope of folding variant narrowing which works on order-sorted unconditional rewrite theories. A calculus that computes answers to reachability problems in membership conditional rewrite theories has been developed and proved correct with respect to idempotent normalized answers.

The work is structured as follows: in Section 2 all needed definitions and properties for rewriting and narrowing are introduced. Section 3 introduces the first part of the narrowing calculus, the one that deals with equational unification. Section 4 introduces the part of the calculus dealing with reachability and its proof of correctness. Section 5 shows the calculus at work. In Section 6, related work, conclusions and current lines of investigation for this work are presented. An extended version of this paper, with all the missing proofs, can be found at `http://maude.sip.ucm.es/cnarrowing/`, together with a previous version of this work with transformation rules and a prototype.

2 Preliminaries

We assume familiarity with rewriting logic [BM06]. There are several language implementations of rewriting logic, including Maude [CDE+07]. Rewriting logic is parameterized by an underlying equational logic. In Maude's case this logic is membership equational logic [Mes97].

2.1 Tower of Hanoi example

Throughout this paper the Tower of Hanoi puzzle will be used as a motivating example to explain the definitions in a less abstract way. We have `Rods` `a`, `b` and `c`, and `Disks` `1`, `2`, `3` and `4` which can slide onto any `Rod`. We call a `Rod` with zero or more stacked `Disks` (written juxtaposed) a `Tower`. If smaller `Disks` are always stacked on top of bigger `Disks` we have a `ValidTower` (abbreviated `VT`). A set of valid towers (written separated by commas) is a `State` (abbreviated `St`). A `move` between a `Pair` of towers (written separated by a $-$ symbol) is defined by the rules: 1) only one `Disk` may be moved at a time, 2) each move consists of taking the upper `Disk` from one `Tower` and placing it on top of another `Tower`, and 3) `Disk` X may be placed on top of `Disk` Y only if X is smaller than Y (written $X < Y = \texttt{t}$, where `t` is the *true* `Boolean` value). The goal of the puzzle is to reach a desired `State` from a given initial `State`.

2.2 Membership equational logic

A *membership equational logic* (MEL) *signature* [BM06] is a triple $\Sigma = (K, \Omega, S)$, with K a set of *kinds*, $\Omega = \{\Sigma_{w;k}\}_{(w;k)\in K^* \times K}$ a many-kinded algebraic signature, and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. For simplicity, we only allow overloading of operators whenever the result belongs in the same kind. The kind of a sort s is denoted by $[s]$. The sets $T_{\Sigma,s}$, $T_\Sigma(X)_s$, $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ denote, respectively, the set of ground Σ-terms with sort s, the set of Σ-terms with sort s over the set X of *sorted* variables, the set of ground Σ-terms with kind k and the set of Σ-terms with kind k over the set X of *sorted* variables. We write T_Σ, $T_\Sigma(X)$ for the corresponding term algebras. $vars(t) \subseteq X$ denotes the set of variables in $t \in T_\Sigma(X)$.

In the Tower of Hanoi puzzle, $\Sigma = (K, \Omega, S)$ is: K=$\{\mathtt{TS}, \mathtt{P}, \mathtt{D}, \mathtt{B}\}$,
Ω=$\{\cdot_{\mathtt{D\ TS;TS}},\ ,_{\mathtt{TS\ TS;TS}},\ -_{\mathtt{TS\ TS;P}},\ \mathtt{move}_{\mathtt{P;P}},\ <_{\mathtt{D\ D;B}}\}$, S=$\{S_{\mathtt{TS}}, S_{\mathtt{P}}, S_{\mathtt{D}}, S_{\mathtt{B}}\}$, $S_{\mathtt{D}}$=$\{\mathtt{Disk}\}$,
$S_{\mathtt{TS}}$=$\{\mathtt{Rod}, \mathtt{VT}, \mathtt{Tower}, \mathtt{St}\}$, $S_{\mathtt{P}}$=$\{\mathtt{Pair}\}$, $S_{\mathtt{D}}$=$\{\mathtt{Disk}\}$, $S_{\mathtt{B}}$=$\{\mathtt{Boolean}\}$.
$\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\{\mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}\}$, and $\{\mathtt{t}\}$ are the *atoms* with sort `Rod`, `Disk`, and `Boolean` respectively.

Positions in a term t: we represent the root of t as ϵ and the other positions as strings of nonzero natural numbers in the usual way, considering t as a tree. The set of positions of a term is written $Pos(t)$. $t|_p$ is the subtree below position p. $t[u]_p$ is the replacement in t of the subterm at position p with term u.

A *substitution* $\sigma : Y \to T_\Sigma(X)$ is a function from a finite set of sorted variables $Y \subseteq X$ to $T_\Sigma(X)$ such that $\sigma(y)$ has the same or lower sort as that of the variable $y \in Y$ ($s_1 \leq s_2$, formally defined in the next paragraph). Substitutions are written as σ=$\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where $Dom(\sigma)$=$\{x_1, \ldots, x_n\}$ and $Ran(\sigma)$=$\bigcup_{i=1}^n vars(t_i)$. The identity substitution is id. The restriction of σ to a set of variables V is $\sigma|_V$. Composition of two substitutions is denoted by $\sigma\sigma'$. For substitutions σ and σ' where $Dom(\sigma) \cap Dom(\sigma')$=$\emptyset$, we denote their union by $\sigma \cup \sigma'$.

A MEL theory [BM06] is a pair $(\Sigma, \mathcal{E})$, where Σ is a MEL signature and $\mathcal{E}$ is a finite set of MEL sentences, either conditional equations or conditional memberships of the forms:

$$(\forall X)\ t{=}t' \ \mathtt{if} \bigwedge_i A_i, \quad (\forall X)\ t{:}s \ \mathtt{if} \bigwedge_i A_i$$

for $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$, the latter stating that t is a term of sort s, provided the condition holds, and each A_i can be of the form $t{=}t'$, $t{:}s$ or $t{:=}t'$ (a *matching* equation). Matching equations are treated as ordinary equations, but they impose a limitation in the syntax of admissible MEL theories, as we will see. We also admit unconditional sentences in $\mathcal{E}$. Order-sorted (*sugared*) notation $s_1 \leq s_2$ can be used instead of $(\forall x{:}[s_1])\ x{:}s_2$ `if` $x{:}s_1$. An operator declaration $f : s_1 \times \cdots \times s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom $(\forall x_1{:}[s_1], \ldots, x_n{:}[s_n])\ f(x_1, \ldots, x_n){:}s$ `if` $\bigwedge_{1\leq i\leq n} x_i{:}s_i$. Given a MEL sentence ϕ, we denote by $\mathcal{E} \vdash \phi$ that ϕ can be deduced from $\mathcal{E}$ using the rules in Figure 1, where $=$ can be either $=$ or $:=$ as explained before [BM12]. The rules of Figure 1 specify a sound and complete calculus. A MEL

$$\frac{t \in T_\Sigma(X)}{(\forall X)t = t}\ \texttt{Reflexivity} \qquad \frac{(\forall X)t = t'}{(\forall X)t' = t}\ \texttt{Symmetry}$$

$$\frac{(\forall X)t_1 = t_2 (\forall X)t_2 = t_3}{(\forall X)t_1 = t_3}\ \texttt{Transitivity} \qquad \frac{(\forall X)t' {:} s \quad (\forall X)t {=} t'}{(\forall X)t {:} s}\ \texttt{Membership}$$

$$\frac{f \in \Sigma_{k_1 \cdots k_n, k} \ (\forall X)t_i {=} t_i' \quad t_i, t_i' {\in} T_\Sigma(X)_{k_i}, 1 {\leq} i {\leq} n}{(\forall X) f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')}\ \texttt{Congruence}$$

$$\frac{((\forall X)\ A_0\ \texttt{if}\ \bigwedge_i A_i) {\in} E \quad \theta {:} X {\to} T_\Sigma(Y) \quad (\forall Y) A_i\theta}{(\forall Y) A_0\theta}\ \texttt{Replacement}$$

Fig. 1. Deduction rules for membership equational logic.

theory $(\Sigma, \mathcal{E})$ has an *initial algebra*, denoted by $T_{\Sigma/\mathcal{E}}$, whose elements are the equivalence classes $[t]_\mathcal{E} \subseteq T_\Sigma$ of ground terms identified by the equations in $\mathcal{E}$.

The MEL theory for the Tower of Hanoi puzzle consists of $\Sigma = (K, \Omega, S)$ and the following set $\mathcal{E}$ of MEL sentences where we omit the universal quantifiers:

$X : \texttt{St}$ *if* $X : \texttt{VT}$; $X : \texttt{Tower}$ *if* $X : \texttt{VT}$; $X : \texttt{St}$ *if* $X : \texttt{Rod}$; $X : \texttt{Tower}$ *if* $X : \texttt{Rod}$;
$X : \texttt{St}$ *if* $X : \texttt{Rod}$; $X : \texttt{VT}$ *if* $X : \texttt{Rod}$; $XY : \texttt{Tower}$ *if* $X : \texttt{Disk} \wedge Y : \texttt{Tower}$;
$X, Y : \texttt{St}$ *if* $X : \texttt{St} \wedge Y : \texttt{St}$; $X, Y = Y, X$; $(X, Y), Z = X, (Y, Z)$;
$X - Y : \texttt{Pair}$ *if* $X : \texttt{Tower} \wedge Y : \texttt{Tower}$; $X - Y = Y - X$;
$X < Y : \texttt{Boolean}$ *if* $X : \texttt{Disk} \wedge Y : \texttt{Disk}$; $XR : \texttt{VT}$ *if* $X : \texttt{Disk} \wedge R : \texttt{Rod}$;
$XYT : \texttt{VT}$ *if* $X : \texttt{Disk} \wedge Y : \texttt{Disk} \wedge T : \texttt{Tower} \wedge X {<} Y = \texttt{t} \wedge YT : \texttt{Vt}$;
$\texttt{1} < \texttt{2} = \texttt{t}$; $\texttt{1} < \texttt{3} = \texttt{t}$; $\texttt{1} < \texttt{4} = \texttt{t}$; $\texttt{2} < \texttt{3} = \texttt{t}$; $\texttt{2} < \texttt{4} = \texttt{t}$; $\texttt{3} < \texttt{4} = \texttt{t}$;
$\texttt{move}(XT - R) = T - XR$ *if* $X : \texttt{Disk} \wedge T : \texttt{Tower} \wedge R : \texttt{Rod}$;
$\texttt{move}(XT - YT') = T - XYT'$ *if* $X : \texttt{Disk} \wedge Y : \texttt{Disk} \wedge T : \texttt{Tower} \wedge$
$\wedge\ T' : \texttt{Tower} \wedge \texttt{X} < \texttt{Y} = \texttt{t}$; $\texttt{move}(X) : \texttt{Pair}$ *if* $X : \texttt{Pair}$.

A single `Disk` stacked on a `Rod` is always a `ValidTower`. For multiple `Disks`, we compare them recursively. The operator `move` distinguishes between two cases: if one `Tower` is empty, i.e. a `Rod`, then we can stack any `Disk` on it; else the sizes of the top `Disks` on each `Tower` must be compared (`<`) and we can stack the smaller one on top of the other.

2.3 Rewriting logic

A rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$ consists of a MEL theory $(\Sigma, \mathcal{E})$ together with a finite set R of *conditional rewrite rules* each of which has the form

$$(\forall X)\ l \to r\ \texttt{if}\ \bigwedge_i p_i {=} q_i \wedge \bigwedge_j w_j {:} s_j \wedge \bigwedge_k l_k \to r_k,$$

where l, r are Σ-terms of the same kind and $=$ can be either $=$ or $:=$. Rewrite rules can also be unconditional.

Such a rewrite rule specifies a *one-step transition* from a state $t[l\theta]_p$ to the state $t[r\theta]_p$, denoted by $t[l\theta]_p \to^1_R t[r\theta]_p$, provided the condition holds. The subterm $t|_p$ is called a *redex*.

$$\frac{t \in T_\Sigma(X)}{(\forall X) t \to t}\ \texttt{Reflexivity} \qquad \frac{(\forall X) t_1 \to t_2,\ (\forall X) t_2 \to t_3}{(\forall X) t_1 \to t_3}\ \texttt{Transitivity}$$

$$\frac{f \in \Sigma_{k_1 \cdots k_n, k}\ (\forall X) t_i \to t_i' \quad t_i, t_i' \in T_\Sigma(X)_{k_i}, 1 \leq i \leq n}{(\forall X) f(t_1, \ldots, t_n) \to f(t_1', \ldots, t_n')}\ \texttt{Congruence}$$

$$((\forall X)\ l \to r\ \texttt{if}\ \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j \wedge \bigwedge_k l_k \to r_k) \in R$$

$$\frac{\theta : X \to T_\Sigma(Y) \quad \bigwedge_i \mathcal{E} \vdash (\forall Y) p_i\theta = q_i\theta \quad \bigwedge_j \mathcal{E} \vdash (\forall Y) w_j\theta : s_j \quad \bigwedge_k (\forall Y) l_k\theta \to r_k\theta}{(\forall Y) l\theta \to r\theta}\ \texttt{Replace}$$

Fig. 2. Deduction rules for rewrite theories.

In the example, R has as only element the conditional rewrite rule: $D, E \to F, G\ if\ D : \texttt{Tower} \wedge E : \texttt{Tower} \wedge F - G := \texttt{move}(D - E) \wedge F : \texttt{Tower} \wedge G : \texttt{Tower}$.

F and G are new variables on the right side of the rule. They are instantiated by matching on the conditional part of the rule.

The inference rules in Figure 2 for rewrite theories can infer all possible computations in the system specified by $\mathcal{R}$ [BM12]. We can *reach* a state v from a state u if we can prove $\mathcal{R} \vdash u \to v$.

The relation $\to^1_{R/\mathcal{E}}$ on $T_\Sigma(X)$ is $=_\mathcal{E} \circ \to^1_R \circ =_\mathcal{E}$. $\to^1_{R/\mathcal{E}}$ on $T_\Sigma(X)$ induces a relation $\to^1_{R/\mathcal{E}}$ on $T_{\Sigma/\mathcal{E}}(X)$, the equivalence relation modulo $\mathcal{E}$, by $[t]_\mathcal{E} \to^1_{R/\mathcal{E}} [t']_\mathcal{E}$ iff $t \to^1_{R/\mathcal{E}} t'$. The transitive (resp. transitive and reflexive) closure of $\to^1_{R/\mathcal{E}}$ is denoted $\to^+_{R/\mathcal{E}}$ (resp. $\to^*_{R/\mathcal{E}}$). We say that a term t is $\to_{R/\mathcal{E}}$*-irreducible* (or just $R/\mathcal{E}$-irreducible) if there is no term t' such that $t \to^1_{R/\mathcal{E}} t'$.

A rewrite rule $l \to r$ `if` *cond*, is *sort-decreasing* if for each substitution σ, we have that for any sort s if $l\sigma \in T_\Sigma(X)_s$ and $(cond)\sigma$ is verified implies $r\sigma \in T_\Sigma(X)_s$. A Σ-equation $t = t'$ is *regular* if $Var(t) = Var(t')$. It is *sort-preserving* if for each substitution σ, we have $t\sigma \in T_\sigma(X)_s$ implies $t'\sigma \in T_\sigma(X)_s$ and vice versa.

A substitution is called $\mathcal{E}$*-normalized* (or normalized) if $x\sigma$ is $\mathcal{E}$-irreducible for all $x \in V$.

The relation $\to^1_{R/\mathcal{E}}$ is *terminating* if there are no infinite rewriting sequences. The relation $\to^1_{R/\mathcal{E}}$ is *operationally terminating* if there are no infinite well-formed proof trees. The relation $\to^1_{R/\mathcal{E}}$ is *confluent* if whenever $t \to^*_{R/\mathcal{E}} t'$ and $t \to^*_{R/\mathcal{E}} t''$, there exists a term t''' such that $t' \to^*_{R/\mathcal{E}} t'''$ and $t'' \to^*_{R/\mathcal{E}} t'''$. In a confluent, terminating, sort-decreasing, membership rewrite theory, for each term $t \in T_\Sigma(X)$, there is a unique (up to $\mathcal{E}$-equivalence) $R/\mathcal{E}$-irreducible term t' obtained by rewriting to *canonical* form, denoted by $t \to^!_{R/\mathcal{E}} t'$, or $t \downarrow_{R/\mathcal{E}}$ when t' is not relevant, which we call $can_{R/\mathcal{E}}(t)$.

2.4 Executable rewrite theories

For a rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$, whether a one step rewrite $t \rightarrow^1_{R/\mathcal{E}} t'$ holds is undecidable in general. We impose additional conditions, similar to those required for functional and system modules in Maude, under which we can decide if $t \rightarrow^1_{R/\mathcal{E}} t'$ holds. We decompose $\mathcal{E}$ into a disjoint union $E \cup A$, with A a set of equational axioms (such as associativity, and/or commutativity, and/or identity). We define the relation $\rightarrow^1_{E,A}$ on $T_\Sigma(X)$ as follows: $t \rightarrow^1_{E,A} t'$ if there is a position $\omega \in Pos(t)$, an equation $l = r$ `if` $cond \in E$, and a substitution σ such that $t|_\omega =_A l\sigma$ (A-matching), $(cond)\sigma$ is satisfied, and $t' = t[r\sigma]_\omega$. The relation $\rightarrow^1_{R,A}$ is similarly defined. We define $\rightarrow^1_{R\cup E,A}$ as $\rightarrow^1_{R,A} \cup \rightarrow^1_{E,A}$. A rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$ is *executable* if each kind k in Σ is nonempty, E, A, and R are finite and the following conditions hold:

1. E and R are *admissible* [CDE+07]. Then we have a *deterministic 3-CTRS* [Ohl02]. Any new variable in the conditions will be instantiated by matching. New variables are distinguished in Maude by using a := symbol instead of = in the condition. They appear on the left terms of these *matching equations*. Conditions in deterministic 3-CTRS's *must* be solved in left to right order.
2. Equality modulo A is decidable and there exists a finite *matching algorithm modulo A*.
3. The equations in E are *sort-decreasing*, and *terminating and confluent modulo A* when we consider them as oriented rules, where $\rightarrow^1_{E/A}$ is defined in the same way as we did for $\rightarrow^1_{R/\mathcal{E}}$.
4. $\rightarrow_{E,A}$ is *coherent with A*, i.e., $\forall t_1, t_2, t_3$ we have $t_1 \rightarrow^+_{E,A} t_2$ and $t_1 =_A t_3$ implies $\exists t_4, t_5$ such that $t_2 \rightarrow^*_{E,A} t_4, t_3 \rightarrow^+_{E,A} t_5$ and $t_4 =_A t_5$ [MT07].

$$\begin{array}{ccc} t_1 & \rightarrow^+_{E,A} & t_2 \\ A & & \\ t_3 & \longrightarrow^+_{E,A} & \end{array} \qquad \begin{array}{ccc} \rightarrow^*_{E,A} & t_4 \\ & A \\ & t_5 \end{array}$$

5. $\rightarrow_{R,A}$ is *$\mathcal{E}$-consistent with A*, i.e., $\forall t_1, t_2, t_3$ we have $t_1 \rightarrow_{R,A} t_2$ and $t_1 =_A t_3$ implies $\exists t_4$ such that $t_3 \rightarrow_{R,A} t_4$ and $t_2 =_\mathcal{E} t_4$. Also $\rightarrow_{R,A}$ is *$\mathcal{E}$-consistent with $\rightarrow_{E,A}$*, i.e., $\forall t_1, t_2, t_3$ we have $t_1 \rightarrow_{R,A} t_2$ and $t_1 \rightarrow^*_{E,A} t_3$ implies $\exists t_4, t_5$ such that $t_3 \rightarrow^*_{E,A} t_4$ and $t_4 \rightarrow_{R,A} t_5$ and $t_2 =_\mathcal{E} t_5$. In both cases the $\rightarrow_{R,A}$ rewriting steps from t_3 and t_4 must be performed with the *same* rule that was applied to t_1 [MT07].

$$\begin{array}{ccc} t_1 & \rightarrow_{R,A} & t_2 \\ A & & \mathcal{E} \\ t_3 & \rightarrow_{R,A} & t_4 \end{array} \qquad \begin{array}{ccccc} t_1 & & \longrightarrow_{R,A} & & t_2 \\ \downarrow^*_{E,A} & & & & \mathcal{E} \\ t_3 & \rightarrow^*_{E,A} & t_4 & \rightarrow_{R,A} & t_5 \end{array}$$

(a) $\mathcal{E}$-consistency of $\rightarrow_{R,A}$ with A (b) $\mathcal{E}$-consistency of $\rightarrow_{R,A}$ with $\rightarrow_{E,A}$

Technically, what coherence means is that the weaker relation $\rightarrow^1_{E,A}$ becomes semantically equivalent to the stronger relation $\rightarrow^1_{E/A}$, so we can decide $t \rightarrow^1_{R/\mathcal{E}}$

t' by finding t'' such that $can_{E,A}(t) \to_R^1 t''$ and $can_{E,A}(t') =_A can_{E,A}(t'')$, which is decidable, since the number of rules is finite and A-matching is decidable and finite.

Under these conditions we can implement $\to_{R/\mathcal{E}}$ on terms using $\to_{R \cup E,A}$ [MT07]. This lemma links $\to_{R/\mathcal{E}}$ with $\to_{E,A}$ and $\to_{R,A}$. Patrick Viry gave a proof for unsorted unconditional rewrite theories [Vir94], which can easily be lifted to our membership conditional case.

Lemma 1. *Let $\mathcal{R} = (\Sigma, \mathcal{E}, R)$ be an executable rewrite theory, that is, it has all the properties specified in Section 2.4. Then $t_1 \to_{R/\mathcal{E}} t_2$ if and only if $t_1 \to_{E,A}^* \to_{R,A} t_3$ for some $t_3 =_{\mathcal{E}} t_2$.*

The rewrite theory for the Tower of Hanoi puzzle is executable if we decompose $\mathcal{E}$ in the following way: the set A has as elements the associative equation and the commutative equations in $\mathcal{E}$; the set E has as elements the rest of equations and all memberships in $\mathcal{E}$, and we add to R the following rule needed for $\mathcal{E}$-consistency:
$D, E, S \to F, G, S \; if \; D : \texttt{Tower} \wedge E : \texttt{Tower} \wedge S : \texttt{State} \wedge \; F - G := \texttt{move}(D - E) \wedge$
$\wedge F : \texttt{Tower} \wedge G : \texttt{Tower}$.

2.5 Unification

Given a rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$, a *$\Sigma$-equation* is an expression of the form $t = t'$ where $t, t' \in T_\Sigma(X)_s$ for an appropriate s. The *$\mathcal{E}$-subsumption* preorder $\ll_{\mathcal{E}}$ on $T_\Sigma(X)_s$ is defined by $t \ll_{\mathcal{E}} t'$ if there is a substitution σ such that $t =_{\mathcal{E}} t'\sigma$. For substitutions σ, ρ and a set of variables V we define $\sigma|_V \ll_{\mathcal{E}} \rho|_V$ if there is a substitution η such that $\sigma|_V =_{\mathcal{E}} (\rho\eta)|_V$. Then we say that ρ is more general than σ with respect to V. When V is not specified, we assume that $V = Dom(\sigma) = Dom(\rho)$ and we say that ρ is more general than σ.

A *system of equations* F is a conjunction of the form $t_1 = t_1' \wedge \ldots \wedge t_n = t_n'$ where for $1 \leq i \leq n$, $t_i = t_i'$ is a Σ-equation. We define $Var(F) = \bigcup_i Var(t_i) \cup Var(t_i')$. An *$\mathcal{E}$-unifier* for F is a substitution σ such that $t_i\sigma =_{\mathcal{E}} t_i'\sigma$ for $1 \leq i \leq n$. For $V = Var(F) \subseteq W$, a set of substitutions $CSU_{\mathcal{E}}^W(F)$ is said to be a *complete set of unifiers modulo $\mathcal{E}$* of F away from W if

- each $\sigma \in CSU_{\mathcal{E}}^W(F)$ is an $\mathcal{E}$-unifier of F;
- for any $\mathcal{E}$-unifier ρ of F there is a $\sigma \in CSU_{\mathcal{E}}^W(F)$ such that $\rho|_V \ll_{\mathcal{E}} \sigma|_V$;
- for all $\sigma \in CSU_{\mathcal{E}}^W(F)$, $Dom(\sigma) \subseteq V$ and $Ran(\sigma) \cap W = \emptyset$.

An $\mathcal{E}$-unification algorithm is *complete* if for any given system of equations it generates a complete set of $\mathcal{E}$-unifiers, which may not be finite. A unification algorithm is said to be *finite* and complete if it terminates after generating a finite and complete set of solutions.

2.6 Reachability goals

Given a rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$, a *reachability goal* G is a conjunction of the form $t_1 \to^* t_1' \wedge \ldots \wedge t_n \to^* t_n'$ where for $1 \leq i \leq n$, $t_i, t_i' \in T_\Sigma(X)_{s_i}$ for

appropriate s_i. We define $Var(G) = \bigcup_i Var(t_i) \cup Var(t'_i)$. A substitution σ is a *solution* of G if $t_i\sigma \rightarrow^*_{R/\mathcal{E}} t'_i\sigma$ for $1 \leq i \leq n$. We define $\mathrm{E}(G)$ to be the system of equations $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$. We say σ is a *trivial solution* of G if it is an $\mathcal{E}$-unifier for $\mathrm{E}(G)$. We say G is trivial if the identity substitution id is a trivial solution of G.

For goals $G : t_1 \rightarrow^* t_2 \wedge \ldots \wedge t_{2n-1} \rightarrow^* t_{2n}$ and $G' : t'_1 \rightarrow^* t'_2 \wedge \ldots \wedge t'_{2n-1} \rightarrow^* t'_{2n}$ we say $G =_{\mathcal{E}} G'$ if $t_i =_{\mathcal{E}} t'_i$ for $1 \leq i \leq 2n$. We say $G \rightarrow_R G'$ if there is an odd i such that $t_i \rightarrow_R t'_i$ and for all $j \neq i$ we have $t_j = t'_j$. That is, G and G' differ only in one subgoal ($t_i \rightarrow t_{i+1}$ vs $t'_i \rightarrow t_{i+1}$), but $t_i \rightarrow t'_i$, so when we rewrite t_i in G to t'_i we get G'. The relation $\rightarrow_{R/\mathcal{E}}$ over goals is defined as $=_{\mathcal{E}} \circ \rightarrow_R \circ =_{\mathcal{E}}$.

2.7 Narrowing

Let t be a Σ-term and W be a set of variables such that $Var(t) \subseteq W$. The R, A-*narrowing* relation on $T_\Sigma(X)$ is defined as follows: $t \rightsquigarrow_{p,\sigma,R,A} t'$ if there is a non-variable position $p \in Pos_\Sigma(t)$, a rule $l \rightarrow r$ `if` *cond* in R, properly renamed, such that $Var(l) \cap W = \emptyset$, and a unifier $\sigma \in CSU_A^{W'}(t|_p = l)$ for $W' = W \cup Var(l)$, such that $t' = (t[r]_p)\sigma$ and $(cond)\sigma$ holds. Similarly E, A-narrowing and $R \cup E, A$-narrowing relations are defined.

2.8 Associated rewrite theory

Any executable Mel theory $(\Sigma, E \cup A)$ has a corresponding rewrite theory $\mathcal{R}_E = (\Sigma', A, R_E)$ associated to it [DLM+08]: we add a fresh new kind *Truth* with a constant tt to Σ, and for each kind $k \in K$ an operator $eq : k\ k \rightarrow Truth$. $\top$ represents a conjunction of any number of tt's. There are rules $eq(x{:}k, x{:}k) \rightarrow tt$ for each kind $k \in K$. For each conditional equation or membership in E the set R_E has a conditional rule or membership of the form

$$t \rightarrow t' \ \textit{if}\ A_1^\bullet \wedge \ldots \wedge A_n^\bullet \qquad t{:}s \ \textit{if}\ A_1^\bullet \wedge \ldots \wedge A_n^\bullet$$

where if A_i is a membership then $A_i^\bullet{=}A_i$, if $A_i \equiv t_i{:=}t'_i$ then $A_i^\bullet$ is $t'_i{\rightarrow}t_i$, and if $A_i \equiv t{=}t'$ then $A_i^\bullet$ is $eq(t, t'){\rightarrow}tt$.

Systems of equations in $(\Sigma, E \cup A)$ with form $G \equiv \bigwedge_{i=1}^m (s_i = t_i)$ become reachability goals in $\mathcal{R}_E$ of the form $\bigwedge_{i=1}^m eq(s_i, t_i) \rightarrow tt$. A substitution σ is a solution of G if there are derivations for $\bigwedge_{i=1}^m (s_i\sigma = t_i\sigma)$, or $\bigwedge_{i=1}^m eq(s_i\sigma, t_i\sigma)$ rewrites to $\top$.

The *inference rules for membership rewriting in* $\mathcal{R}_E$ are the ones in Figure 3, adapted from [DLM+08, Fig. 4, p. 12], where the rules are defined for context-sensitive membership rewriting.

3 Conditional narrowing modulo unification

Narrowing allows us to assign values to variables in such a way that a reachability goal holds. We implement narrowing using a calculus that has the following properties:

$$\frac{t_1 \to^1 t_2,\ t_2 \to t_3}{t_1 \to t_3}\ \texttt{Transitivity} \qquad \frac{t \to^1 t', t' : s}{t : s}\ \texttt{Subject Reduction}$$

$$\frac{t =_A t'}{t \to t'}\ \texttt{Reflexivity} \qquad \frac{t_i \to^1 t_i'}{f(t_1, \ldots, t_i, \ldots, t_n) \to^1 f(t_1, \ldots, t_i', \ldots, t_n)}\ \texttt{Congruence}$$

$$\frac{t \to t' \text{ if } A_1^\bullet \ldots A_n^\bullet \in \mathcal{R}_E \text{ and } u =_A t\sigma \quad A_1^\bullet\sigma \ldots A_n^\bullet\sigma}{u \to^1 t'\sigma}\ \texttt{Replacement}$$

$$\frac{t : s \text{ if } A_1^\bullet \ldots A_n^\bullet \in \mathcal{R}_E \text{ and } u =_A t\sigma \quad A_1^\bullet\sigma \ldots A_n^\bullet\sigma}{u : s}\ \texttt{Membership}$$

Fig. 3. Inference rules for membership rewriting.

1. If σ is an $R/\mathcal{E}$-normalized idempotent solution for a reachability goal G, the calculus can compute a more general answer $\sigma \ll_{\mathcal{E}} \sigma'$ for G.
2. If the calculus computes an answer σ for G, then σ is a solution for G.

That is, we want to compute a complete set of answers for G, a set that includes a generalization of any possible solution for G, with respect to $R/\mathcal{E}$-normalized substitutions.

We are going to split this task into two subtasks: first we will solve the part of the calculus that deals with unification; second, we will solve the part that deals with reachability.

3.1 Calculus rules for unification

We assume we are working with an executable rewrite theory named M. We refer to the set of equations and memberships in M as E, to the set of rules as R and to the set of axioms as A. We also assume that we have an A-unification algorithm that returns a *CSU* for any pair of terms.

A unification equation is a term $s{:}S = t{:}T$, which is a shorthand for the system of equations $s = t \wedge s = X_S \wedge t = Y_T$ (we will also write $s = t$, $s{:}S$, $t{:}T$). This means that we intend to unify s and t, with resulting sorts S and T respectively. A unification goal is a sequence (understood as conjunction) of unification equations.

Admissible goals, or simply goals, are any sequence of $s{:}S{=}t{:}T$, $s{:}S{:=}t{:}T$, $s{:}S{\to}t{:}T$, $s{:}S{\to^1}t{:}T$ and $t{:}T$. Any condition in an equation, of the form $s{=}t$ or $s{:=}t$ is turned into an admissible goal by adding inferred sorts to it. If any term s is a variable or a constant, we use the sort of s as inferred sort. If the term is of the form $f(\bar{s})$, we use the kind of any membership for f.

Our calculus is defined by the following set of inference rules derived from those in Figure 3. The first two rules, $[u]$ and $[x]$, transform *equational* problems into *rewriting* problems modulo axioms, rule $[u]$ playing the part of the added rules $eq(x{:}k, x{:}k) \to tt$ in the associated rewrite theory; rule $[n]$ describes one step of unification narrowing where the conditions on the applied rule are turned

into subgoals and the instantiated right side of the rule ($r\theta$) is required to have a sort which is a common subsort of S and T; rule $[t]$ allows us to apply several unification narrowing steps; rule $[i]$ decomposes a term allowing rule $[n]$ to be applied to any subterm of it; rule $[r]$ allows instantiation of variables on unificable terms; rule $[m1]$ solves the membership problem for variables, and rules $[s]$ and $[m2]$ for the rest of terms, using the membership conditions in E:

- $[u]$ *unification*

$$\frac{s{:}S = t{:}T, G'}{s{:}S' \to X_{S'}{:}S', t{:}S' \to X_{S'}{:}S', G'}$$

where $X_{S'}$ fresh variable, $S' \leq S, S' \leq T$.

- $[x]$ *matching*

$$\frac{s{:}S := t{:}T, G'}{t{:}S' \to s{:}S', G'}$$

where $S' \leq S, S' \leq T$.

- $[n]$ *narrowing*

$$\frac{s{:}S \to^1 X{:}T, G'}{((c,)X{:}S', G')\rho\theta}$$

where s is not a variable, $(c)eq\ l{=}r\ (if\ c) \in E$ has fresh variables, $S' \leq S, S' \leq T$, $\theta \in CSU_A(s = l)$, $\rho{=}\{X \mapsto r\}$.

- $[t]$ *transitivity*

$$\frac{s{:}S \to t{:}T, G'}{s{:}S' \to^1 X_{S'}{:}S', X_{S'}{:}S' \to t{:}S', G'}$$

where $X_{S'}$ fresh variable, $S' \leq S, S' \leq T$.

- $[i]$ *imitation*

$$\frac{f(\bar{s}{:}\bar{S}){:}S \to^1 X{:}T, G'}{G'\theta, s_i{:}S_i \to^1 X'_{S_i}{:}S_i, X\theta{:}S', G''\theta}$$

with $X \notin Var(s), \theta = \{X \mapsto f((s_1, \ldots, s_{i-1}, X'_{S_i}{:}S_i, s_{i+1}, \ldots, s_n))\}$, X'_{S_i} fresh variable, $S' \leq S, S' \leq T$.

- $[r]$ *removal of equations*

$$\frac{s{:}S \to t{:}T, G'}{(G', s{:}S', G')\theta}$$

with $\theta \in CSU_A(s = t)$, $S' \leq S, S' \leq T$

- $[s]$ *subject reduction*

$$\frac{s{:}S, G'}{s{:}[S] \to^1 X_S{:}S, G'}$$

X_S fresh variable.

– [$m1$] *membership*

$$\frac{X_S{:}T, G'}{(G')\theta}$$

where $\theta = \{X_S \mapsto X'_{S'}\}$ with $X'_{S'}$ fresh variable and $S' \leq S, S' \leq T$.

– [$m2$] *membership*

$$\frac{s{:}S, G'}{((c,)\, G')\theta}$$

where $(c)mb\ t{:}T$ (*if* c) is a fresh variant, with $T \leq S$, of a (conditional) membership in E, and $\theta \in CSU_A(s = t)$.

From a unification equation u a derivation is made applying rules of the calculus. If the derivation ends in the empty goal, denoted by $\square$, then the composition of the substitutions used on each derivation step, restricted to those variables appearing in u, is a computed answer for u.

Theorem 1. *The calculus for unification is sound and weakly complete.*

That is, given a unification goal G, if $G \rightsquigarrow^*_\sigma \square$ then $G\sigma$ can be derived, so σ is a solution for G in $\rightarrow_{E/A}$, and if ρ is an E/A-normalized idempotent answer of G ($G\rho \rightarrow^*_{E/A} \top$), then there is ρ' idempotent, with $\rho \ll_A \rho'$, such that $G \rightsquigarrow_{\rho'} \square$.

4 Reachability by conditional narrowing

Conditional narrowing relies on conditional unification. As we have used the symbol $\rightarrow$ in the calculus rules for unification, we will use a different symbol $\Rightarrow$ in the calculus rules for reachability. Our goal, given a reachability problem $\bigwedge_i s_i{:}S_i \Rightarrow t_i{:}T_i$, is to find a solution σ (ground or not) such that $\bigwedge_i s_i\sigma{:}S_i \Rightarrow_{R/\mathcal{E}} t_i\sigma{:}T_i$. For executable rewrite theories this is equivalent to $\bigwedge_i s_i\sigma{:}S_i \Rightarrow_{R\cup E,A} \bigwedge_i t_i\sigma{:}T_i$. These new calculus rules deal with the $\rightsquigarrow_{R,A}$ part. Narrowing, we call it *replacement* here, takes place only at position ϵ of terms, thanks to new transitivity and imitation calculus rules.

Reachability goals are any sequence (understood as conjunction) of subgoals of the form $s{:}S \Rightarrow t{:}T$. Admissible goals, or simply goals, are now extended to be any sequence of $s{:}S \Rightarrow t{:}T$, $s{:}S \Rightarrow^1 t{:}T$, $s{:}S = t{:}T$, $s{:}S \rightarrow t{:}T$, $s{:}S \rightarrow^1 t{:}T$, $s{:}S \rightarrow^1 t{:}T$, $s{:}S := t{:}T$ and $t{:}T$. If the calculus derives the empty goal from a reachability goal G with a substitution σ, then σ is a computed answer for G.

As for unification, any reachability subgoal in our calculus of the form of $s{:}S \Rightarrow^{(1)} t{:}T$ is equivalent to the admissible goal $s \Rightarrow^{(1)} t$, $s{:}S$, $t{:}T$.

4.1 Calculus rules for reachability

Reachability by conditional narrowing is achieved using the calculus rules presented in Section 3, extended with the following calculus rules, based on the deduction rules for rewrite theories in Figure 2. Rule [X] solves reachability problems by unification; rule [R] applies one step of reachability narrowing; rule

$[T]$ enables reachability narrowing modulo and multiple steps of reachability narrowing. It is a direct consequence a 1 ; rule $[I]$ allows us to imitate narrowing at non root term positions, replacing the rewriting rule for congruence, that can now be achieved by transitivity and imitation. Recall that narrowing steps for reachability ($\Rightarrow^1$), which are generated by rule $[T]$, impose no sort within the given kind on the right side of the step:

- $[X]$ *reflexivity*

$$\frac{s{:}S \Rightarrow t{:}T, G'}{s{:}S = t{:}T, G'}$$

- $[R]$ *replacement*

$$\frac{s{:}S \Rightarrow^1 X_{[S]}{:}[S], G'}{(s{:}S, (c,), G')\rho\theta}$$

where s is not a variable, $(c)rl\ l \Rightarrow r\ (if\ c)$ is a fresh variant of a (conditional) rule in R, $\rho = \{X_{[S]} \mapsto r\}$, $\theta \in CSU_A(s = l)$.

- $[T]$ *transitivity*

$$\frac{s{:}S \Rightarrow t{:}T, G'}{s{:}S \rightarrow X'_S{:}S, X'_S{:}S \Rightarrow^1 X''_{[S]}{:}[S], X''_{[S]}{:}[S] \Rightarrow t{:}T, G'}$$

where X'_S and $X''_{[S]}$ are fresh variables.

- $[I]$ *imitation*

$$\frac{f(\bar{s}{:}\bar{S}){:}S \Rightarrow^1 X_{[S]}{:}[S], G'}{s_i{:}S_i \Rightarrow^1 X'_{S_i}{:}S_i, f(\bar{s}{:}\bar{S}){:}S, G'\theta}$$

where $X_{[S]} \notin vars(s), \theta = \{X_{[S]} \mapsto f((s_1, \ldots, X'_{S_i}{:}S_i, \ldots, s_n))\}$, X'_{S_i} fresh variable.

From a reachability goal r a derivation is made applying rules of the calculus. Each application of the *reflexivity* rule generates a unification equation. These unification equations as well as any generated membership goals must be solved using the calculus rules for unification. If the derivation ends with an *empty goal*, written $\square$, then the composition of the substitutions used on each derivation step, restricted to those variables appearing in r, is a computed answer for r.

Theorem 2. *The calculus for reachability is sound and weakly complete.*

That is, given a reachability goal G, if $G \rightsquigarrow^*_\sigma \square$ then $G\sigma$ can be derived, so σ is a solution for G in $\rightarrow_{R/\mathcal{E}}$, and if θ is an $R/\mathcal{E}$-normalized idempotent answer for a reachability problem G in $\rightarrow_{R/\mathcal{E}}$, then there is σ idempotent, with $\theta \ll_{\mathcal{E}} \sigma$, such that $G \rightsquigarrow^*_\sigma \square$.

Proof. We prove correctness of the calculus for reachability with respect to $R/\mathcal{E}$-normalized (equivalently $R \cup E, A$) idempotent substitutions for the executable rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$ in $\rightarrow_{R/\mathcal{E}}$.

1. Soundness: By structural induction on the calculus rule for reachability applied.

2. Completeness: We prove that for $R/\mathcal{E}$-normalized idempotent answers $\Rightarrow^1$ solves $\rightarrow^1_{R,A}$ reachability problems and $\Rightarrow$ solves $\rightarrow^*_{R/\mathcal{E}}$ reachability problems, according to [MT07, Theorem 3] and Lemma 1. Then it follows that if θ is an $R/\mathcal{E}$-normalized idempotent answer for a reachability problem G in $\rightarrow_{R/\mathcal{E}}$, then there is σ idempotent, with $\theta \ll_{\mathcal{E}} \sigma$, such that $G \rightsquigarrow^*_\sigma \square$. Inferred sorts are treated as in the proof of completeness of the calculus for unification (see extended version). We don't show the inferred sorts here.
   (a) We prove that if $s\rho \rightarrow^1_{R,A} t$ then $s \Rightarrow^1 t' \rightsquigarrow^*_\sigma \square$, with $\rho \ll_{\mathcal{E}} \sigma$ and $t \ll_{\mathcal{E}} t'$. By definition there is a position p in $s\rho$, a rule $l \rightarrow r$ *if* $c \in R$ and a matching θ such that $s\rho|_p = l\theta$, $c\theta$ can be derived and $t \equiv (s\rho)[r\theta]_p$. By the same reasoning we used for the completeness of the calculus for unification, p must be a nonvariable position in s. Otherwise ρ would not be $R/\mathcal{E}$-normalized. From $s \Rightarrow^1 X$, by imitation we can reach position p, turning our reachability problem into $s|_p \Rightarrow^1 X^p$ with $\eta = \{X \mapsto s[X^p]_p\}$. Applying replacement, as $s\rho|_p = l\theta$, there is $\sigma (\equiv \rho' \cup \theta') \in CSU_A(s\rho|_p = l)$, with $\rho \ll_{\mathcal{E}} \rho'$, $\theta \ll_{\mathcal{E}} \theta'$ and $t' \equiv X\eta\sigma \equiv (s\rho')[r\theta']_p$.
   It is important to remember, again, that ACU-coherence completion allows A-unification of the left term of the ACU-coherence completed version of the rule, l, with the whole $s\rho|_p$ whenever the original left term l can be A-unified with some subterm of a recombination of $s\rho|_p$.
   (b) We prove that if $s\rho \rightarrow^*_{R/\mathcal{E}} t\rho$, ρ is a solution, then $s \Rightarrow t \rightsquigarrow^*_\sigma \square$, with $\rho \ll_{\mathcal{E}} \sigma$. We distinguish two cases:
      – Reflexive case: $s\rho =_{\mathcal{E}} t\rho$. Then $s \Rightarrow t \rightsquigarrow_{[X]} s = t \rightsquigarrow^*_\sigma \square$, with $\rho \ll_{\mathcal{E}} \sigma$ by correctness of the calculus for unification.
      – Rest of the cases. According to [MT07, Lemmas 7 and 8] and the Lemma in Section 2.4 it suffices to show that $(\rightsquigarrow^*_{E,A} \rightsquigarrow_{R,A})^+ =_{\mathcal{E}}$ is implemented by $\Rightarrow$. This is done in the transitivity rule

$$\frac{s{:}S \Rightarrow t{:}T, G'}{s{:}S \rightarrow X'_S{:}S, X'_S{:}S \Rightarrow^1 X''_{[S]}{:}[S], X''_{[S]}{:}[S] \Rightarrow t{:}T, G'}$$

   $s{:}S \rightarrow X'_S$ implements $\rightsquigarrow^*_{E,A}$ as proved in the calculus for unification. $X'_S{:}S \Rightarrow^1 X''_{[S]}{:}[S]$ implements $\rightsquigarrow_{R,A}$ as proved in the previous point. $X''_{[S]}{:}[S] \Rightarrow t{:}T$ allows iteration (the $^+$ part) through several uses of the transitivity rule ending with the $=_{\mathcal{E}}$ part through the use of the reflexivity rule, which is the only rule that enables us to exit the loop generated by the transitivity rule.
   Finally, correct typing is ensured because $s{:}S$ and $t{:}T$ are included as conditions.

5 Example

As an example of our calculus we use the specification of the Tower of Hanoi puzzle in Section 2 and the reachability problem $(3T^0_T, b, c){:}S \Rightarrow (a, b, T^1_T){:}S$, where from a `State` composed of one `Tower` with `Disk` 3 on top of it and two `Towers`

with `Rods` b and c alone respectively we want to reach a `State` composed of two `Towers` with `Rods` a and b alone respectively and another `Tower`. The subindex of each variable means its type (sort or kind) and we write D, R, V, T, P, S instead of `Disk`, `Rod`, `ValidT`, `Tower`, `Pair`, `State` for readability.:

1. $\underline{(3T^0_T, b, c){:}S \Rightarrow (a, b, T^1_T){:}S} \rightsquigarrow_{[T]}$
   Transitivity decomposes reachability into several rewriting narrowing steps.
2. $\underline{(3T^0_T, b, c){:}S \to X^1_S{:}S}, X^1_S{:}S \Rightarrow^1 X^2_{[S]}{:}[S], X^2_{[S]}{:}[S] \Rightarrow (a, b, T^1_T){:}S$
   $\rightsquigarrow_{[r],\{T^0_T \mapsto a, X^1_S \mapsto (3a,b,c)\}} T^0_T$ is instantiated through rule $[r]$.
3. $\underline{(3a, b, c){:}S}, (3a, b, c){:}S \Rightarrow^1 X^2_{[S]}{:}[S], X^2_{[S]}{:}[S] \Rightarrow (a, b, T^1_T){:}S$
   We focus on the first subgoal.
4. $\underline{(3a, b, c){:}S} \rightsquigarrow_{[m2], S^1_{[S]}, S^2_{[S]}{:}S \ \mathit{if}\ S^1_{[S]}{:}S \wedge S^2_{[S]}{:}S, \{S^1_{[S]} \mapsto (3a,b), S^2_{[S]} \mapsto c\}}$
5. $\underline{c{:}S}, (3a, b){:}S \rightsquigarrow \ldots$
6. $\underline{3a{:}S} \rightsquigarrow_{[m2], X_{[D]} R_{[R]}{:}V \ \mathit{if}\ X_{[D]}{:}D \wedge R_{[R]}{:}R, \{X_{[D]} \mapsto 3, R_{[R]} \mapsto a\}}$. OK because $V \leq S$.
7. $\underline{3{:}D, a{:}R} \rightsquigarrow \ldots$ similar to previous steps. First subgoal finished.
8. $(3a, b, c){:}S \Rightarrow^1 X^2_{[S]}{:}[S], X^2_{[S]}{:}[S] \Rightarrow (a, b, T^1_T){:}S$. We focus on the first subgoal.
9. $\underline{(3a, b, c){:}S \Rightarrow^1 X^2_{[S]}{:}[S]} \rightsquigarrow_{[R], D_{[T]}, E_{[T]}, X_{[S]} \to F_{[T]}, G_{[T]}, X_{[S]} \ \mathit{if}}$
   $D_{[T]}{:}T \wedge E_{[T]}{:}T \wedge X_{[S]}{:}S \wedge F_{[T]}{:}T \wedge G_{[T]}{:}T \wedge F_{[T]} - G_{[T]} := \mathit{move}(D_{[T]} - E_{[T]}),$
   $\theta = \{D_{[T]} \mapsto 3a, E_{[T]} \mapsto c, X_{[S]} \mapsto b\}, \rho = \{X^2_{[S]}{:}[S] \mapsto F_{[T]}, G_{[T]}, X_{[S]}\}$ Narrowing step.
10. $\underline{(3a, b, c){:}S, 3a{:}T, c{:}T, b{:}S}, (F_{[T]} - G_{[T]}){:}[P] := \mathit{move}(3a - c){:}[P] \rightsquigarrow \ldots$
11. $\underline{F_{[T]} - G_{[T]}{:}[P] := \mathit{move}(3a - c){:}[P]} \rightsquigarrow_{[x]}$
12. $\underline{\mathit{move}(3a - c){:}[P] \to F_{[T]} - G_{[T]}{:}[P]} \rightsquigarrow_{[t]}$
    Transitivity decomposes unification into several unification narrowing steps.
13. $\underline{\mathit{move}(3a - c){:}[P] \to^1 Y_{[P]}{:}[P], Y_{[P]}{:}[P] \to F_{[T]} - G_{[T]}{:}[P]} \rightsquigarrow_{[n],}$
    $\mathit{move}(X_{[D]} T_{[T]} - R_{[R]}) = T_{[T]} - X_{[D]} R_{[R]} \ \mathit{if}\ X_{[D]}{:}D \wedge T_{[T]}{:}T \wedge R_{[R]}{:}R,$
    $\theta = \{X_{[D]} \mapsto 3, T_{[T]} \mapsto a, R_{[R]} \mapsto c\}, \rho = \{Y_{[P]} \mapsto T_{[T]} - X_{[D]} R_{[R]}\}$
    Unification narrowing step. $Y_{[P]}$ is instantiated to a ground term.
14. $\underline{a - 3c{:}[P], 3{:}[D], a{:}[T], c{:}[R]}, a - 3c{:}[P] \to F_{[T]} - G_{[T]}{:}[P] \rightsquigarrow \ldots$
15. $\underline{a - 3c{:}[P] \to F_{[T]} - G_{[T]}{:}[P]} \rightsquigarrow_{[r], \theta_1 = \{F_{[T]} \mapsto a, G_{[T]} \mapsto 3c\}}$ Removal of equations.
16. $\underline{a - 3c{:}[P]} \rightsquigarrow \ldots$ We omit this and go back to the second subgoal on step 8.
17. $\underline{(a, 3c, b) : [S] \Rightarrow (a, b, T^1_T){:}S} \rightsquigarrow_{[X]} \ldots$
18. $\underline{(a, 3c, b) : S \to X_S{:}S}, (a, b, T^1_T){:}S \to X_S{:}S \rightsquigarrow_{[r], \{X_S \mapsto (a,3c,b)\}}$
19. $\underline{(a, 3c, b) : S}, (a, b, T^1_T){:}S \to (a, 3c, b){:}S \rightsquigarrow \ldots$
20. $\underline{(a, b, T^1_T){:}S \to (a, 3c, b){:}S} \rightsquigarrow_{[r], \{T^1_T \mapsto 3c\}} T^1_T$ is instantiated through rule $[r]$.
21. $\underline{(a, b, 3c) : S} \rightsquigarrow \ldots \square$

From the substitutions in steps 2 and 20 the answer $\{T^1_T \mapsto 3c, T^0_T \mapsto a\}$ is computed. The calculus has found the solution $(3a, b, c){:}S \Rightarrow (a, b, 3c){:}S$ which is an instance of the given reachability problem $(3T^0_T, b, c){:}S \Rightarrow (a, b, T^1_T){:}S$.

6 Related work, conclusions and future work

A classic reference in equational conditional narrowing modulo is the work of Bockmayr [Boc93]. The topic is addressed here for Church-Rosser equational CTRS with empty axioms, but non terminating axioms (like ACU) are not allowed. Non conditional narrowing modulo order-sorted equational logics is covered by Meseguer and Thati [MT07], the reference for recent development in this area, which is actively being used for cryptographic protocol analysis. This work is partially based on the work of Viry [Vir94] where $R/\mathcal{E}$ rewriting is defined in terms of R, A and E, A for unsorted rewrite theories. Another topic addressed by the present work, membership equational logic, is defined by Meseguer [Mes97]. An equivalent rewrite system for MEL theories is presented by Durán, Lucas et al. [DLM+08], allowing unification by rewriting. Strategies, which also play a main role in narrowing, have been studied by Antoy, Echahed and Hanus [AEH94]. Their needed narrowing strategy, for inductively sequential rewrite systems, generates only narrowing steps leading to a computed answer. Recently Escobar, Sasse and Meseguer [ESM12] have developed the concepts of variant and folding variant, a narrowing strategy for order-sorted unconditional rewrite theories that terminates on those theories having the *finite variant property*. As an extension to rewrite theories Bruni and Meseguer [BM06] have defined *generalized rewrite theories* that support context-sensitive rewriting, thus allowing rewrites only on certain positions of terms.

In this work we have developed a narrowing calculus for unification in membership equational logic and a narrowing calculus for reachability in rewrite theories with an underlying membership equational logic. The main features in these calculi are that they make use of membership information whenever possible, reducing the state space, and also that they only allow steps leading to a different state, no mutual cancelling steps are allowed. The calculi have been proved correct. This work is part of a bigger effort where we attempt to explore the possibilities of performing conditional narrowing with constraint solvers. A transformation for rules and goals that will make both calculi strongly complete is under study. Strong completeness of reachability for topmost rewrite theories, Russian dolls configurations and linear theories are also under study. Finally, decidability of the calculus for unification in the case of *operationally terminating* [LM09] MEL theories with a finitary and complete A-unification algorithm, using the required strategy for deterministic 3-CTRS's of solving subgoals from left to right, is being studied.

Our current line of investigation also intends to study the extension of the calculi to handle constraints and their connection with external constraint solvers for domains such as finite domains, integers, Boolean values, etc., that could greatly improve the performance of any implementation. We also plan on the extension of the calculi, adding support for generalized rewrite theories. Better strategies that may help reducing the state space will also be studied. All the improvements will have new sets of transformation rules that will allow their implementation on Maude.

References

[AEH94] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In H.-J. Boehm, B. Lang, and D. M. Yellin, editors, *POPL*, pages 268–279. ACM Press, 1994.

[BM06] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[BM12] K. Bae and J. Meseguer. Model checking LTLR formulas under localized fairness. In F. Durn, editor, *WRLA*, volume 7571 of *Lecture Notes in Computer Science*, pages 99–117. Springer, 2012.

[Boc93] A. Bockmayr. Conditional narrowing modulo a set of equations. *Applicable Algebra in Engineering, Communication and Computing*, 4:147–168, 1993.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *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, July 2007.

[CM96] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. *Electronic Notes in Theoretical Computer Science*, 4:126–148, 1996.

[DLM+08] F. Durán, S. Lucas, C. Marché, J. Meseguer, and X. Urbain. Proving operational termination of membership equational programs. *Higher-Order and Symbolic Computation*, 21(1-2):59–88, 2008.

[DM12] F. Durán and J. Meseguer. On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *The Journal of Logic and Algebraic Programming*, 81(7-8):816–850, 2012.

[EMM05] S. Escobar, C. Meadows, and J. Meseguer. A rewriting-based inference system for the NRL protocol analyzer: grammar generation. In V. Atluri, P. Samarati, R. Küsters, and J. C. Mitchell, editors, *FMSE*, pages 1–12. ACM, 2005.

[ESM12] S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. *The Journal of Logic and Algebraic Programming*, 81(7-8):898–928, 2012.

[Fay78] M.J. Fay. *First-order Unification in an Equational Theory.* University of California, 1978.

[LM09] S. Lucas and J. Meseguer. Operational termination of membership equational programs: the order-sorted way. *Electr. Notes Theor. Comput. Sci.*, 238(3):207–225, 2009.

[Mes90] J. Meseguer. Rewriting as a unified model of concurrency. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90 Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 384–400. Springer, 1990.

[Mes97] J. Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *WADT*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.

[MT07] J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.

[Ohl02] E. Ohlebusch. *Advanced topics in term rewriting.* Springer, 2002.

[Vir94] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis, D. G. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE*, volume 817 of *Lecture Notes in Computer Science*, pages 648–660. Springer, 1994.

Language Definitions as Rewrite Theories

Andrei Arusoaie[1], Dorel Lucanu[1], Vlad Rusu[2], Traian-Florin Şerbănuţă[1,3], Andrei Ştefănescu[4], and Grigore Roşu[4]

[1] Alexandru Ioan Cuza University, Iaşi, Romania
[2] Inria Lille Nord Europe, France
[3] University of Bucharest, Romania
[4] University of Illinois at Urbana-Champaign, USA

Abstract. $\mathbb{K}$ is a formal framework for defining the operational semantics of programming languages. It includes software tools for compiling $\mathbb{K}$ language definitions to Maude rewrite theories, for executing programs in the defined languages based on the Maude rewriting engine, and for analyzing programs by adapting various Maude analysis tools. A recent extension to the $\mathbb{K}$ tool suite is an automatic transformation of language definitions that enables the symbolic execution of programs, i.e., the execution of programs with symbolic inputs. In this paper we investigate the theoretical relationships between $\mathbb{K}$ language definitions and their translations to Maude, between symbolic extensions of $\mathbb{K}$ definitions and their Maude encodings, and how the relations between $\mathbb{K}$ definitions and their symbolic extensions are reflected on their respective representations in Maude. These results show, in particular, how analyses performed with Maude tools can be formally lifted up to the original language definitions.

1 Introduction

$\mathbb{K}$ [11] is a framework for formally defining the semantics of programming languages. The current version of $\mathbb{K}$ includes options that have Maude [3] as a backend: the $\mathbb{K}$ compiler transforms any $\mathbb{K}$ definition into a Maude module; then, the $\mathbb{K}$ runner uses Maude to run or analyze programs in the defined language.

Recently, $\mathbb{K}$ has been extended with symbolic execution support [2]. Briefly, a $\mathbb{K}$ language definition is automatically transformed into a *symbolic-language* definition, such that the concrete executions of programs using the symbolic definition are symbolic executions of programs using the original language definition. The transformation amounts to incorporating *path conditions* in program configurations, and to changing the language's semantic rules so that they match on *symbolic configurations* and that they automatically update the path conditions.

Symbolic executions are called *feasible* if their path conditions are satisfiable. Two results relating concrete and symbolic program executions are proved in [2]: *coverage*, saying that for each concrete execution there is a feasible symbolic one taking the same path on the program; and *precision*, saying that for each feasible symbolic execution there is a concrete one taking the same program path.

In this paper we propose two ways of representing $\mathbb{K}$ language definitions in Maude: a *faithful* representation and an *approximate* one. We then study

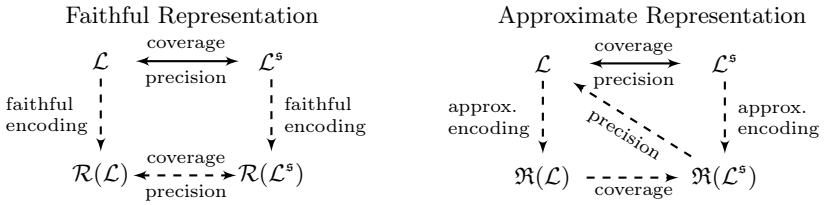


Fig. 1. Faithful vs. Approximate representations

the relationships between $\mathbb{K}$ language definitions (including the symbolic ones, obtained by the above-described transformation) and their representations in Maude. We also show how the coverage and precision results, which relate a language $\mathcal{L}$ and its symbolic extension $\mathcal{L}^{\mathfrak{s}}$, are reflected on their respective representations in Maude. These results show, in particular, how (symbolic) analyses performed with Maude tools on the (faithful and approximate) Maude representations of languages can be lifted up to the original language definitions. The various results that we have obtained are graphically depicted in the diagrams from Figure 1 (dashed arrows show the results proved in the paper). In the faithful encoding, each semantic rule of the language definition $\mathcal{L}$ is translated into a rewrite rule of the rewrite theory $\mathcal{R}(\mathcal{L})$. Equations are only introduced in order to express equality in the data domain. The resulting rewrite theory is proved to be *executable* by Maude, and the transition system generated by the language definition is shown to be isomorphic to the one generated by the rewrite theory. Some variations of this encoding are also discussed, all of which satisfy the executability and faithfulness properties. As a consequence, both positive and negative results of reachability analyses, obtained on rewrite theories (i.e., by using the Maude *search* command) also hold on the original language definitions. Moreover, all symbolic reachability analysis results obtained on the rewrite-theory representation $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ of a symbolic language $\mathcal{L}^{\mathfrak{s}}$ also hold on the rewrite-theory representation $\mathcal{R}(\mathcal{L})$ of the language $\mathcal{L}$. The latter property is analoguous to the results obtained in [10], where *rewriting modulo SMT* is shown to be related to (usual) rewriting in a *sound* and *complete* way.

For nontrivial language definitions, the faithful encoding is not very practical, because it typically generates a huge state-space that is not amenable to reachability analysis. This is why we introduce approximate representations of language definitions as *two-layered rewrite theories*. These approximations are obtained by splitting the semantic rules of the language into two sets, called *layers*, such that the first layer forms a terminating rewrite system. The one-step rewriting in such a theory is obtained by computing an irreducible form w.r.t. rules from the first layer (according to a given strategy), and then applying a rule from the second layer.

In an (approximating) two-layered rewrite theory $\mathfrak{R}(\mathcal{L})$, only a subset of the executions of programs in the original language $\mathcal{L}$ are represented. The consequence is that only positive results of reachability analyses on the two-layered rewrite theories can be lifted up to the corresponding language definitions. In addition, to reduce the state-space to be explored, the approximate encoding of

a language by a two-layered rewrite theory can also be seen as the output of a *compiler* that solves some semantic choices left by the language definition at compile-time. For example, in C, the order in which the operands of addition are evaluated is a compile-time choice. By turning the operand-evaluation rules into first-layer rules, and by letting Maude automatically execute these rules in various orders according to certain strategies, one can reproduce the various design compile-time choices for the evaluation of arguments.

We note that approximating two-layered rewrite theories have some limitations: only the coverage property relating the language definition $\mathcal{L}$ to its symbolic version $\mathcal{L}^{\mathfrak{s}}$ also holds on their respective approximate encodings theories; the precision property holds only in some restricted cases. However, the precision property between the approximate symbolic encoding $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ and the language definition $\mathcal{L}$ always holds. Hence, one can trace symbolic reachability analyses (performed on $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$) back to programs in $\mathcal{L}$, and also (in some restricted cases) to the representation of programs in $\mathfrak{R}(\mathcal{L})$, which, as discussed above, can be seen as compiled programs where some semantic choices are left to the compiler.

Organisation. In Section 2 we present our working examples, which are two programs belonging to the CinK kernel of C++, which was specified in $\mathbb{K}$ [7]. A partial description of the $\mathbb{K}$ definition for CinK is included. In Section 3 we introduce a formal notion of a language-definition framework, which allows us to make our approach independent of the $\mathbb{K}$ language definitional framework and to abstract away some particular implementation details of $\mathbb{K}$. For the same reason, we will be using rewrite theories (instead of their implementations as Maude modules) for the encodings of language definitions. We also briefly present the language-independent symbolic execution approach [2] and recap some essential notions related to the executability of rewrite theories.

Section 4 presents the faithful and the approximate representations of language definitions into a rewrite theory and the various relations between them (graphically depicted in Figure 1). Section 5 presents the applications of these representations to the compilation of $\mathbb{K}$ language definitions as Maude modules. Finally, Section 6 presents conclusions and related work.

2 Running Example

Our running example is CinK [7], a kernel of the C++ programming language. The $\mathbb{K}$ definition of CinK can be found on the $\mathbb{K}$ Framework Github repository: `http://github.com/kframework/cink-semantics`. As any $\mathbb{K}$ definition, it consists of the language syntax, given using a BNF-style grammar, and of its semantics, given using rewrite rules on configurations. In this paper we only exhibit a small part of the $\mathbb{K}$ definition of CinK, whose syntax is shown in Figure 2. Some of the grammar productions are annotated with $\mathbb{K}$-specific attributes.

A major feature of C++ expressions is that given by the "sequenced before" relation [1], which defines a partial order over the evaluation of subexpressions. This can be easily expressed in $\mathbb{K}$ using the *strict* attribute to specify an evaluation order for an operation's operands. If the operator is annotated with the

$$
\begin{array}{lll}
\mathit{Exp} & ::= \mathit{Id} \mid \mathit{Int} & \\
& \mid \texttt{++}\ \mathit{Exp} & [\mathit{strict}, \mathit{prefinc}] \\
& \mid \texttt{--}\ \mathit{Exp} & [\mathit{strict}, \mathit{predec}] \\
& \mid \mathit{Exp}\ \texttt{/}\ \mathit{Exp} & [\mathit{strict}(\mathit{all}(\mathit{context}(\mathit{rvalue}))), \mathit{divide}] \\
& \mid \mathit{Exp}\ \texttt{+}\ \mathit{Exp} & [\mathit{strict}(\mathit{all}(\mathit{context}(\mathit{rvalue}))), \mathit{plus}] \\
& \mid \mathit{Exp}\ \texttt{>}\ \mathit{Exp} & [\mathit{strict}(\mathit{all}(\mathit{context}(\mathit{rvalue})))] \\
\mathit{Stmt} & ::= \mathit{Exps}\ \texttt{;} & [\mathit{strict}] \\
& \mid \{\mathit{Stmts}\} & \\
& \mid \texttt{while}\ (\mathit{Exp})\mathit{Stmt} & \\
& \mid \texttt{return}\ \mathit{Exp}\ \texttt{;} & [\mathit{strict}(\mathit{all}(\mathit{context}(\mathit{rvalue})))] \\
& \mid \texttt{if}\ (\mathit{Exp})\mathit{Stmt}\ \texttt{else}\ \mathit{Stmt} & [\mathit{strict}(1(\mathit{context}(\mathit{rvalue})))]
\end{array}
$$

Fig. 2. CinK syntax

strict attribute then its operands will be evaluated in a nondeterministic order. For instance, all the binary operations are strict. Hence, they may induce non-determinism in programs because of possible side-effects in their arguments.

Another feature is given by the classification of expressions into *rvalues* and *lvalues*. For instance, in the expression `x = x + 2` the first occurrence of `x` is an lvalue whereas the second one is an rvalue. The arguments of binary operations are evaluated as rvalues and their results are also rvalues, while, e.g., both the argument of the prefix-increment operation and its result are lvalues. The *strict* attribute for such operations has a sub-attribute *context* for wrapping any subexpression that must be evaluated as an rvalue. Other attributes ($\mathit{funcall}, \mathit{divide}, \mathit{plus}, \mathit{minus}, \ldots$) are names associated to each syntactic production, which can be used to refer them.

The $\mathbb{K}$ framework uses *configurations* to store program states. A configuration is a nested structure of cells, which typically include the program to be executed, I/O streams, values for program variables, and other additional information. The configuration of CinK (Figure 3) includes the $\langle\rangle_{\mathsf{k}}$ cell containing the code that remains to be executed, which is represented as a list of computation tasks $C_1 \curvearrowright C_2 \curvearrowright \ldots$ to be executed in the given order. Computation tasks are typically statements and expression evaluations. The memory is modeled using two cells $\langle\rangle_{\mathsf{env}}$ (which holds a map from variables to addresses) and $\langle\rangle_{\mathsf{state}}$ (which holds a map from addresses to values). The configuration also includes a cell for the function call stack $\langle\rangle_{\mathsf{stack}}$ and another one $\langle\rangle_{\mathsf{return}}$ for the return values of functions.

$$\langle\ \langle \$PGM \rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\ \ \langle\cdot\rangle_{\mathsf{store}}\ \ \langle\cdot\rangle_{\mathsf{stack}}\ \ \langle\cdot\rangle_{\mathsf{return}}\ \ \rangle_{\mathsf{cfg}}$$

Fig. 3. CinK configuration

When the configuration is initialised at runtime, a CinK program is loaded in the $\langle\rangle_{\mathsf{k}}$ cell, and all the other cells remain empty. A $\mathbb{K}$ *rule* is a topmost rewrite rule specifying transitions between configurations. Since usually only a small part of the configuration is changed by a rule, a *configuration abstraction* mechanism is used, allowing one to only specify the parts transformed by the rule. For instance, the (abstract) rule for addition, shown in Figure 4, represents the (concrete) rule

$$\langle\langle I_1\ \texttt{+}\ I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\rangle_{\mathsf{cfg}}$$
$$\Rightarrow$$
$$\langle\langle I_1 +_{\mathit{Int}} I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\rangle_{\mathsf{cfg}}$$

I_1:*Int* + I_2:*Int* ⇒ $I_1 +_{Int} I_2$ [*plus*]
I_1:*Int* / I_2:*Int* ⇒ $I_1 /_{Int} I_2$ **requires** $I_2 \neq_{Int} 0$ [*division*]
`if(` *true* `)` *St*:*Stmt* `else _` ⇒ *St* [*if-true*]
`if(` *false* `) _ else` *St*:*Stmt* ⇒ *St* [*if-false*]
`while(` *B*:*Exp* `)` *St*:*Stmt* ⇒ `if(` *B* `){` *St* `while(` *B* `)` *St* `else {}}` [*while*]
V:*Val* `;` ⇒ · [*instr-expr*]
⟨`++lval(` *L*:*Loc* `)` ⇒ `lval(` *L* `)` ⋯⟩$_{\mathsf{k}}$⟨⋯ *L* ↦ (*V*:*Int* ⇒ *V* $+_{Int}$ 1) ⋯⟩$_{\mathsf{store}}$ [*inc*, *memw*]
⟨`--lval(` *L*:*Loc* `)` ⇒ `lval(` *L* `)` ⋯⟩$_{\mathsf{k}}$⟨⋯ *L* ↦ (*V*:*Int* ⇒ *V* $-_{Int}$ 1) ⋯⟩$_{\mathsf{store}}$ [*dec*, *memw*]
⟨⟨`lval(` *L*:*Loc* `)`= *V*:*Val* ⇒ *V* ⋯⟩$_{\mathsf{k}}$⟨⋯ *L* ↦ _ ⇒ *V* ⋯⟩$_{\mathsf{store}}$ ⋯⟩$_{\mathsf{cfg}}$ [*update*, *memw*]
⟨⟨`$lookup(` *L*:*Loc* `)` ⇒ *V* ⋯⟩$_{\mathsf{k}}$⟨⋯ *L* ↦ *V*:*Val* ⋯⟩$_{\mathsf{store}}$ ⋯⟩$_{\mathsf{cfg}}$ [*lookup*, *memr*]
`{` *Sts*:*Stmts* `}` ⇒ *Sts* [*block*]

Fig. 4. Subset of rules from the K semantics of CinK

where $+_{Int}$ is the mathematical operation for addition. Note that the ellipses in a cell (e.g., ⟨ ⋯⟩$_{\mathsf{k}}$) represent the part of the cell not affected by the rule.

The rule for division has a side condition which restricts its application. The conditional statement `if` has two corresponding rules, one for each possible evaluation of the condition expression. The rule for the `while` loop performs an unrolling into an `if` statement. The increment and update rules have side effects in the ⟨⟩$_{\mathsf{store}}$ cell, modifying the value stored at a specific address. Finally, the reading of a value from the memory is specified by the lookup rule, which matches a value in the ⟨⟩$_{\mathsf{store}}$ and places it in the ⟨⟩$_{\mathsf{k}}$ cell. The auxiliary construct `$lookup` is used, e.g., when a program variable is evaluated as an rvalue.

In addition to these rules (written by the $\mathbb{K}$ user), the $\mathbb{K}$ framework automatically generates so-called *heating* and *cooling* rules, which are induced by *strict* attributes. We show only the case of division, which is strict in both arguments:

$$A_1 \text{ / } A_2 \Rightarrow \mathit{rvalue}(A_1) \curvearrowright \square \text{ / } A_2 \quad (1) \qquad \mathit{rvalue}(I_1) \curvearrowright \square \text{ / } A_2 \Rightarrow I_1 \text{ / } A_2 \quad (3)$$
$$A_1 \text{ / } A_2 \Rightarrow \mathit{rvalue}(A_2) \curvearrowright A_1 \text{ / } \square \quad (2) \qquad \mathit{rvalue}(I_2) \curvearrowright A_1 \text{ / } \square \Rightarrow A_1 \text{ / } I_2 \quad (4)$$

where $\square$ is a special symbol, destined to receive the result of an evaluation.

We shall be using the following two programs in the sequel. The program `counter` in Figure 5 is nondeterministic; nondeterminism arises from the undefined evaluation order for the arguments of the + operation and from the side-effects in its arguments. The program `log` in the same figure is a symbolic one

```
int counter = 1;
int inc() {
  return ++counter;
}
int dec() {
  return --counter;
}
int main() {
  return  inc() + dec();
}
```

a) The program `counter`

```
int main() {
int k, x;
  x = A:Int;  //A:Int is a symbolic value
  k = 0;
  while (x > 0) {
    ++k;
    x = x / 2;
  }
}
```

b) The program `log`

Fig. 5. Two C++ programs

because `A:Int` is a symbolic value, which can denote any integer value. When it is completed the variable `k` holds $\lfloor \log_2(A) \rfloor$ where $\lfloor \_ \rfloor$ denotes the integer part of a real number. In Section 5 we show how the behaviours of these programs can be analysed using our encodings of the CinK language as Maude programs.

3 Background

3.1 The Ingredients of a Language Definition

In this section we identify the ingredients of language definitions in an algebraic and term-rewriting setting. The concepts are explained on the $\mathbb{K}$ definition of CinK. We assume the reader is familiar with the basics of algebraic specification and rewriting. A language $\mathcal{L}$ can be defined as a triple $(\Sigma, \mathcal{T}, \mathcal{S})$, consisting of:

1. A many-sorted algebraic signature Σ, which includes at least a sort *Cfg* for *configurations* and a sort *Bool* for *constraint formulas*. For the sake of presentation, we assume in this paper that the constraint formulas are Boolean terms built with a subsignature $\Sigma^{\mathsf{Bool}} \subseteq \Sigma$ including the Boolean constants and operations. Σ may also include other subsignatures for other data sorts, depending on the language $\mathcal{L}$ (e.g., integers, identifiers, lists, maps, ...). Let Σ^{Data} denote the subsignature of Σ consisting of all *data* sorts and their operations. We assume that the sort *Cfg* and the syntax of $\mathcal{L}$ are not data, i.e., they are defined in $\Sigma \setminus \Sigma^{\mathsf{Data}}$. Let T_Σ denote the Σ-algebra of ground terms and $T_{\Sigma,s}$ denote the set of ground terms of sort s. Given a sort-wise infinite set of variables *Var*, let $T_\Sigma(Var)$ denote the free Σ-algebra of terms with variables, $T_{\Sigma,s}(Var)$ denote the set of terms of sort s with variables, and $var(t)$ denote the set of variables occurring in the term t.
2. A Σ^{Data}-model $\mathcal{D}$, which interprets the data sorts and operations. For convenience, we assume that $\mathcal{D}_d \subset \Sigma_d$ for each data sort d, i.e., the constants are elements of the corresponding signature. Let $\mathcal{T} \triangleq \mathcal{T}(\mathcal{D})$ denote the free Σ-model generated by $\mathcal{D}$. The satisfaction relation $\rho \models b$ between valuations ρ and constraint formulas $b \in T_{\Sigma,Bool}(Var)$ is defined by $\rho \models b$ iff $\rho(b) = \mathcal{D}_{true}$. For simplicity, we write $true, false, 0, 1 \ldots$ instead of $\mathcal{D}_{true}, \mathcal{D}_{false}, \mathcal{D}_0, \mathcal{D}_1, \ldots$.
3. A set $\mathcal{S}$ of rewrite rules. Each rule is a pair of the form $l \wedge b \Rightarrow r$, where $l, r \in T_{\Sigma,Cfg}(Var)$ are the rule's *left-hand-side* and *right-hand-side*, respectively, and $b \in T_{\Sigma,Bool}(Var)$ is the *condition*. The formal definitions for rules and for the transition system defined by them are given below.

Remark 1. For the sake of presentation, here we consider only "pure" language definitions, where the semantics is given only by semantic rules between configurations. Some definitions may include additional functions defined by equations. For such cases the language definition may additionally include a set of axioms A_0, e.g., associativity and/or commutativity of some functions, and a set of equations E_0. Then the model $\mathcal{T}$ is the free algebra modulo $A_0 \cup E_0$. We believe that the approach presented in this paper can be extended to these more involved definitions, but this requires more investigation and is left for future work.

We now formally introduce the notions required for defining semantic rules.

Definition 1 (pattern [12]). *A* pattern *is an expression of the form* $\pi \wedge b$*, where* $\pi \in T_{\Sigma,Cfg}(Var)$ *is a* basic pattern *and* $b \in T_{\Sigma,Bool}(Var)$*. If* $\gamma \in T_{Cfg}$ *and* $\rho : Var \to \mathcal{T}$ *then we write* $(\gamma, \rho) \models \pi \wedge b$ *iff* $\gamma = \rho(\pi)$ *and* $\rho \models b$*.*

A basic pattern π defines a set of (concrete) configurations, and the condition b gives additional constraints these configurations must satisfy.

Remark 2. The above definition is a particular case of a definition in [12]. There, a pattern is a first-order logic (FOL) formula with configuration terms as subformulas. In this paper we keep the conjunction notation from FOL but separate basic patterns from constraints. Note that FOL formulas can be encoded as terms of sort *Bool*, where the quantifiers become constructors. The satisfaction relation $\models$ is then defined, for such terms, like the usual FOL satisfaction.

We identify basic patterns π with patterns $\pi \wedge \mathit{true}$. Sample patterns are $\langle\langle I_1 + I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\langle\langle I_1 \,/\, I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0$.

Definition 2 (rule, transition system). *A* rule *is a pair of patterns of the form* $l \wedge b \Rightarrow r$ *(note that* r *is in fact the pattern* $r \wedge \mathit{true}$*). Any set* $\mathcal{S}$ *of rules defines a labelled transition system* $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}})$ *such that* $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ *iff there exist* $\alpha \triangleq (l \wedge b \Rightarrow r) \in \mathcal{S}$ *and* $\rho : Var \to \mathcal{T}$ *such that* $(\gamma, \rho) \models l \wedge b$ *and* $(\gamma', \rho) \models r$*.*

We write $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ when there exists $\alpha \in \mathcal{S}$ such that $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$, and denote by $\Rightarrow_{\mathcal{S}}^{+}$ the transitive closure of $\Rightarrow_{\mathcal{S}}$.

3.2 Symbolic Execution

We briefly recap our approach to symbolic execution from [2]. The main idea is to automatically generate a new definition $(\Sigma^{\mathfrak{s}}, \mathcal{T}^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$ for a language $\mathcal{L}^{\mathfrak{s}}$ from a given definition $(\Sigma, \mathcal{T}, \mathcal{S})$ of a language $\mathcal{L}$. The new language $\mathcal{L}^{\mathfrak{s}}$ has the same syntax, and its semantics extends $\mathcal{L}$'s data domains with symbolic values and adapts the semantical rules of $\mathcal{L}$ to deal with the new domains.

Let $V^{\mathfrak{s}}$ denote an infinite, data sort-wise set of *symbolic values*, disjoint from Var and from symbols in Σ. The data algebra is extended to $\mathcal{D}^{\mathfrak{s}}$, which is the algebra of ground terms over the signature $\Sigma^{\mathsf{Data}}(V^{\mathfrak{s}})$. The signature $\Sigma^{\mathfrak{s}}$ extends Σ with the symbolic values $V^{\mathfrak{s}}$ as constants, a new sort $Cfg^{\mathfrak{s}}$ and a constructor $\_ \wedge \_ : Cfg \times Bool \to Cfg^{\mathfrak{s}}$. The model $\mathcal{T}^{\mathfrak{s}}$ is defined as being the free $\Sigma^{\mathfrak{s}}$-model generated by $\mathcal{D}^{\mathfrak{s}}$, similarly to how $\mathcal{T}$ is built over $\mathcal{D}$. The ground terms $\pi \wedge \phi \in \mathcal{T}^{\mathfrak{s}}_{Cfg^{\mathfrak{s}}}$ are called *symbolic configurations*. Let $[\![\pi \wedge \phi]\!]$ denote the set of concrete configurations $\{\gamma \mid (\exists \rho)\, (\gamma, \rho) \models \pi \wedge \phi\}$.

Thanks to the rule transformation procedure presented in [2], we make without loss of generality the assumption that the basic patterns in left-hand sides of rules do not contain operations on data, and the rules are left-linear. Concrete semantic rules $l \wedge b \Rightarrow r \in \mathcal{S}$ are then systematically transformed into rules

$$l \wedge \psi \Rightarrow r \wedge (\psi \wedge b) \tag{5}$$

where $\psi \in Var$ is a fresh variable of sort *Bool* playing the role of a path condition. This means that symbolic rules are applied like concrete rules, except for the fact that the current path condition ψ is enriched with the rule's condition b.

Then, the symbolic execution of $\mathcal{L}$ programs is the concrete execution of the corresponding $\mathcal{L}^{\mathfrak{s}}$ programs, i.e., the application of the rewrite rules in the semantics of $\mathcal{L}^{\mathfrak{s}}$. Building the definition of $\mathcal{L}^{\mathfrak{s}}$ amounts to extending the signature Σ to a symbolic signature $\Sigma^{\mathfrak{s}}$, extending the Σ-algebra $\mathcal{T}$ to a $\Sigma^{\mathfrak{s}}$-algebra $\mathcal{T}^{\mathfrak{s}}$, and turning the concrete rules $\mathcal{S}$ into symbolic rules $\mathcal{S}^{\mathfrak{s}}$. The transition system $(\mathcal{T}^{\mathfrak{s}}_{Cfg^{\mathfrak{s}}}, \Rightarrow_{\mathcal{S}^{\mathfrak{s}}})$ is defined using Definitions 1, 2 applied to $\mathcal{L}^{\mathfrak{s}}$. In [2] it is proved that the symbolic transition system forward-simulates the concrete one, and that the concrete transition system backward-simulates the symbolic one. These two results then imply the naturally expected properties of symbolic execution.

Theorem 1 (Coverage [2]). *For every concrete execution* $\gamma_0 \overset{\alpha_1}{\Longrightarrow}_{\mathcal{S}} \gamma_1 \overset{\alpha_2}{\Longrightarrow}_{\mathcal{S}} \cdots \overset{\alpha_n}{\Longrightarrow}_{\mathcal{S}} \gamma_n \overset{\alpha_{n+1}}{\Longrightarrow}_{\mathcal{S}} \cdots$ *there is a symbolic execution* $\pi_0 \wedge \phi_0 \overset{\alpha_1}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_1 \wedge \phi_1 \overset{\alpha_2}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \cdots \overset{\alpha_n}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_n \wedge \phi_n \overset{\alpha_{n+1}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \cdots$ *such that* $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ *for* $i = 0, 1, \ldots$.

A symbolic configuration $\pi \wedge \phi \in \mathcal{T}^{\mathfrak{s}}_{Cfg^{\mathfrak{s}}}$ is *satisfiable* if there is a valuation $\vartheta : V^{\mathfrak{s}} \to \mathcal{D}$ such that $\vartheta \models \phi$ (which is equivalent to $[\![\pi \wedge \phi]\!] \neq \emptyset$). We call a symbolic execution *feasible* if all its configurations are satisfiable.

Theorem 2 (Precision [2]). *For every feasible symbolic execution* $\pi_0 \wedge \phi_0 \overset{\alpha_1}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_1 \wedge \phi_1 \overset{\alpha_2}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \cdots \overset{\alpha_n}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_n \wedge \phi_n \overset{\alpha_{n+1}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \cdots$ *there is a concrete execution* $\gamma_0 \overset{\alpha_1}{\Longrightarrow}_{\mathcal{S}} \gamma_1 \overset{\alpha_2}{\Longrightarrow}_{\mathcal{S}} \cdots \overset{\alpha_n}{\Longrightarrow}_{\mathcal{S}} \gamma_n \overset{\alpha_{n+1}}{\Longrightarrow}_{\mathcal{S}} \cdots$ *such that* $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ *for* $i = 0, 1, \ldots$.

3.3 Rewrite Theories

A rewrite theory [3] $\mathcal{R} = (\Sigma, E \cup A, R)$ consists of a signature Σ, a set of equations E, a set of axioms A, e.g., associativity, commutativity, identity or combinations of these, and a set of rewrite rules R of the form $l \to r$ **if** b, where l and r are terms with variables and b is a term of sort *Bool*. We are only interested in rewrite theories $\mathcal{R}$ that are *executable* [9], i.e., $(\Sigma, E \cup A, R)$ where:

1. there exists a matching algorithm modulo A;
2. $(\Sigma, E \cup A)$ is ground Church-Rosser and terminating modulo A (the equations E are seen here as rewrite rules oriented from left to right). Thus, each ground term t has a canonical form $can_{E/A}(t)$ that is unique modulo the axioms A;
3. R is *ground coherent w.r.t. E modulo A* [13]: for all $t, t_1 \in T_\Sigma$ with $t \to_{R/A} t_1$ there is $t_2 \in T_\Sigma$ s.t. $can_{E/A}(t) \to_{R/A} t_2$ and $can_{E/A}(t_1) =_A can_{E/A}(t_2)$.

The relation $\to_{R/A}$ denotes the one-step rewriting relation defined by applying a rule from R modulo axioms A: $u \to_{R/A} v$ iff there are terms u', v', a rule $l \to r$ **if** b in R, position p in u', and substitution σ such that $u =_A u'$, $v =_A v'$, $u'|_p = \sigma(l)$, $v' = u[\sigma(r)]_p$, and $\sigma(b) =_A true$. We use $t|_p$ to denote the subterm of t at position p, and $t[u]_p$ to denote the term obtained from t by replacing the subterm at position p with u.

The *rewriting relation* $\to_{\mathcal{R}}$ defined by an executable rewrite theory $\mathcal{R}$ is: $t_1 \to_{\mathcal{R}} t_2$ iff $can_{E/A}(t_1) \to_{R/A} t_2'$ and $can_{E/A}(t_2') = t_2$. This is equivalent to $\to_{R/(E \cup A)}$ due to confluence and coherence. We write $t_1 \overset{\alpha}{\to}_{\mathcal{R}} t_2$ to emphasise that $\alpha \triangleq (l \to r \text{ if } b) \in R$ is applied in the rewriting step $can_{E/A}(t_1) \to_{R/A} t_2'$.

4 Translating Language Definitions into Rewrite Theories

This section includes the main contribution of the paper. We introduce two encodings of language definitions as rewrite theories: a faithful encoding and an approximate encoding. Since the symbolic extension of a language is also a language definition, we automatically get encodings of both concrete languages and their symbolic extensions. We investigate how the properties relating a language definition and its symbolic extension are reflected on their respective encodings.

4.1 Faithful Encoding

Definition 3 (faithful encoding). *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition. The* faithful encoding *of $\mathcal{L}$ is $\mathcal{R}(\mathcal{L}) = (\Sigma, E \cup A, R)$, where*

- *$A = \emptyset$;*
- *for each operation f in Σ^{Data} and $d_1, \ldots, d_n \in \mathcal{D}$ of corresponding sorts, E includes an equation $f(d_1, \ldots, d_n) = \mathcal{D}_f(d_1, \ldots, d_n)$;*
- *$R = \mathcal{S}$, where each rule $l \wedge b \Rightarrow r \in \mathcal{S}$ becomes a rewrite rule $l \rightarrow r$ **if** $b \in R$.*

Theorem 3. *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition. Then $\mathcal{R}(\mathcal{L})$ is an executable rewrite theory satisfying $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ iff $\gamma \overset{\alpha}{\rightarrow}_{\mathcal{R}(\mathcal{L})} \gamma'$, for all $\gamma, \gamma' \in \mathcal{T}_{Cfg}$.*

Remark 3. The construction of the rewrite theory $\mathcal{R}(\mathcal{L})$, with data domain $\mathcal{D} \subseteq \Sigma^{\mathsf{Data}}$ defined by the set of equations E given in Definition 3, corresponds to the data domains $\mathcal{D}$ being *builtin sorts* in the Maude terminology. A builtin sort is a sort that is not built algebraically but one that, for efficiency reasons, is directly implemented in code (C++ code in the case of Maude). For example, natural numbers are specified by the equational specification $0 : \mathsf{Nat}, s : \mathsf{Nat} \rightarrow \mathsf{Nat}$, but using the resulting unary-notation for them would be highly inefficient. This is why natural numbers are implemented as builtins. The construction $\mathcal{R}(\mathcal{L})$ can, however, be extended to accomodate non-builtin sorts, i.e., sorts that are defined as the *initial model* of a finite set of equations E' that are confluent and terminating modulo a set A of axioms. For this, it is enough to ensure that $E' \cup E$ is also confluent and terminating modulo A - where E is the set of equations given in the proof of Theorem 3. This typically happens, as E and E' refer to different sorts - the builtin ones for the former, and the non-builtin ones for the latter. If this is the case then the proof of the ground coherence property in Theorem 3 still holds, because it only depends on $E' \cup E$ being confluent and terminating modulo A, not on the particular form of the equations. The proof of faithfulness of the encoding remains the same. This observation is important, since it ensures that we obtain executable Maude rewrite-theories $\mathcal{R}(\mathcal{L})$ for languages-definitions $\mathcal{L}$ whose data are specified using either builtin sorts or non-builtin sorts. The faithfulness of the encoding then ensures that all results of reachability analyses (either positive or negative) performed on $\mathcal{R}(\mathcal{L})$, e.g., obtained using Maude's *search* command, also hold on $\mathcal{L}$.

The symbolic extension of a language definition can be encoded as a rewrite theory as well. Let $\mathcal{L}^{\mathfrak{s}} = (\Sigma^{\mathfrak{s}}, \mathcal{T}^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$ be the symbolic extension of $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$.

Recall that $\Sigma^{\mathfrak{s}}$ is Σ extended with the constructor of symbolic configurations $\_ \wedge \_$ and with the symbolic values $V^{\mathfrak{s}}$ seen as constants. The symbolic configurations are ground terms $\pi \wedge \phi \in \mathcal{T}^{\mathfrak{s}}_{Cfg^{\mathfrak{s}}}$. If $\mathcal{R}(\mathcal{L}^{\mathfrak{s}}) = (\Sigma^{\mathfrak{s}}, E \cup A, R)$ is the faithful encoding given by Theorem 3, then $E = A = \emptyset$ because the data algebra $\mathcal{D}^{\mathfrak{s}}$ we considered is the $\Sigma^{\mathsf{Data}}(V^{\mathfrak{s}})$-algebra of the ground terms built over $\mathcal{D}$ and $V^{\mathfrak{s}}$. Recall that we assumed that $\mathcal{D} \subseteq \Sigma \subseteq \Sigma^{\mathsf{Data}}(V^{\mathfrak{s}})$.

The relationship between a language definition $\mathcal{L}$ and its symbolic extension $\mathcal{L}^{\mathfrak{s}}$ can be now reflected at the level of the encodings $\mathcal{R}(\mathcal{L})$ and $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$. A symbolic configuration $\pi \wedge \phi$ consists of a configuration ground term π (of sort *Cfg*) and a formula ground term ϕ (of sort *Bool*). The constants $V^{\mathfrak{s}}$ play the role of logical variables, and the definition of satisfiability for patterns extends to their representations as symbolic configurations. Moreover, the notion of feasible execution in $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ is defined similarly to how it is defined for $\mathcal{L}^{\mathfrak{s}}$. The following two results are direct consequences of Theorems 3, 1, and 2.

Corollary 1 (Coverage for Encoding Rewrite Theories). *For every concrete execution* $\gamma_0 \xrightarrow{\alpha_0}_{\mathcal{R}(\mathcal{L})} \gamma_1 \xrightarrow{\alpha_2}_{\mathcal{R}(\mathcal{L})} \cdots \xrightarrow{\alpha_n}_{\mathcal{R}(\mathcal{L})} \gamma_n \xrightarrow{\alpha_{n+1}}_{\mathcal{R}(\mathcal{L})} \cdots$ *there is a symbolic execution* $\pi_0 \wedge \phi_0 \xrightarrow{\alpha_1}_{\mathcal{R}(\mathcal{L}^{\mathfrak{s}})} \pi_1 \wedge \phi_1 \xrightarrow{\alpha_2}_{\mathcal{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \xrightarrow{\alpha_n}_{\mathcal{R}(\mathcal{L})} \pi_n \wedge \phi_n \xrightarrow{\alpha_{n+1}}_{\mathcal{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$ *such that* $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ *for* $i = 0, 1, \ldots$.

Corollary 2 (Precision for Encoding Rewrite Theories). *For every feasible symbolic execution* $\pi_0 \wedge \phi_0 \xrightarrow{\alpha_1}_{\mathcal{R}(\mathcal{L}^{\mathfrak{s}})} \pi_1 \wedge \phi_1 \xrightarrow{\alpha_2}_{\mathcal{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \xrightarrow{\alpha_n}_{\mathcal{R}(\mathcal{L})} \pi_n \wedge \phi_n \xrightarrow{\alpha_{n+1}}_{\mathcal{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$ *there is a concrete execution* $\gamma_0 \xrightarrow{\alpha_0}_{\mathcal{R}(\mathcal{L})} \gamma_1 \xrightarrow{\alpha_2}_{\mathcal{R}(\mathcal{L})} \cdots \xrightarrow{\alpha_n}_{\mathcal{R}(\mathcal{L})} \gamma_n \xrightarrow{\alpha_{n+1}}_{\mathcal{R}(\mathcal{L})} \cdots$ *such that* $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ *for* $i = 0, 1, \ldots$.

The faithful encoding thus enjoys nice theoretical properties, but it has a limited practical value when we consider actual $\mathbb{K}$ definitions of nontrivial languages:

- The heating and cooling rules, which are symmetric to each other, may lead to infinite rewritings;
- The generated state space may be very large, even for small programs.

Therefore in the next section we introduce a new kind of rewrite theories that under-approximate $\mathbb{K}$ definitions in order to be more efficient in practice.

4.2 Approximate Encoding

There are currently two proposals for obtaining abstractions of the rewrite theories: equational abstraction [9] or transforming some semantical rules into equations [6]. The former amounts to basically deriving a new definition, where the new model $\mathcal{T}$ is the quotient of the original one, usually requiring substantial input from the user, which is something we would like to avoid.

The latter might not be suitable for language definitions in general because, semantically, it would equate elements that are supposed to be distinct in $\mathcal{T}$. Consider a language construct `randBool` with two rules: `randBool => true` and `randBool => false`. Assume now we want to analyze a program which uses `randBool`, but who fails to satisfy a given property regardless of whether

`randBool` transits to `true` or to `false`. In this case it might beneficial to collapse the state space by considering only one of the cases; however, if we transform the two rules above into equations, this will semantically identify `true` and `false` in $\mathcal{T}$, collapsing much more of the state space than desirable. An additional operational concern is that transforming certain rules into equations might destroy coherence and/or confluence, thus falling out of the executability requirements.

Two-layered rewrite theories, introduced below, allow us to preserve the benefits of the techniques above (state space reduction, efficient execution), while avoiding their semantical consequences (unnecessary collapse of states in the semantical model $\mathcal{T}$).

Definition 4. *A* two-layered rewrite theory *is a tuple* $\mathfrak{R} = (\Sigma, E \cup A, 1R \cup 2R, \varepsilon)$*, where* $(\Sigma, E \cup A, 1R \cup 2R)$ *is an executable rewrite theory,* $E \cup 1R$ *is ground terminating modulo* A*, and* $\varepsilon : T_\Sigma \to T_\Sigma$ *is a function that, for any* $t \in T_\Sigma$*, returns an element in the set of* $(E \cup 1R)/A$*-irreducible terms* $\{t' \in T_\Sigma \mid t \to^!_{(E \cup 1R)/A} t'\}$ *(which is nonempty precisely because* $E \cup 1R$ *is ground terminating modulo* A*). The one-step rewrite relation* $\twoheadrightarrow_{\mathcal{R}}$ *is defined by* $t_1 \twoheadrightarrow_{\mathcal{R}} t_2$ *iff* $\varepsilon(t_1) \to_{2R/A} t'_2$ *and* $can_{E/A}(t'_2) =_A t_2$*.*

Examples of two-layered rewrite theories are shown in Section 5.

Theorem 4. *Let* $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ *be a language definition and* $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, 1R \cup 2R, \varepsilon)$ *be a two-layered rewrite theory with* $(\Sigma, E \cup A, 1R \cup 2R)$ *built as in Definition 3 but where the set of rules is partitioned into two subsets* $1R$ *and* $2R$ *and* $E \cup 1R$ *is terminating modulo* A*. If* $\gamma \twoheadrightarrow_{\mathfrak{R}(\mathcal{L})} \gamma'$ *then* $\gamma \Rightarrow^+_{\mathcal{S}} \gamma'$*.*
We say that $\mathfrak{R}(\mathcal{L})$ *is an* approximate encoding *of* $\mathcal{L}$*.*

Corollary 3 (precision for approximate encoding). *Let* $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ *be a language definition and* $\mathfrak{R}(\mathcal{L}^\mathfrak{s}) = (\Sigma, E \cup A, 1R \cup 2R, \varepsilon)$ *be an approximate encoding of* $\mathcal{L}^\mathfrak{s}$*. For each feasible symbolic execution* $\pi_0 \wedge \phi_0 \to_{\mathcal{R}^\mathfrak{s}} \pi_1 \wedge \phi_1 \to_{\mathfrak{R}(\mathcal{L}^\mathfrak{s})} \cdots \to_{\mathfrak{R}(\mathcal{L}^\mathfrak{s})} \pi_n \wedge \phi_n \to_{\mathfrak{R}(\mathcal{L}^\mathfrak{s})} \cdots$ *there is a concrete execution in* $\mathcal{L}$*:* $\gamma_0 \Longrightarrow^+_{\mathcal{S}} \gamma_1 \Longrightarrow^+_{\mathcal{S}} \cdots \Longrightarrow^+_{\mathcal{S}} \gamma_n \Longrightarrow^+_{\mathcal{S}} \cdots$ *such that* $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ *for* $i = 0, 1, \ldots$*.*

An interesting and practically relevant question is whether the coverage/precision relationships between $\mathcal{L}$ and $\mathcal{L}^\mathfrak{s}$ can be reflected on the level of the approximate encodings as two-layered rewrite theories. To investigate these relationships, we have to find a way to define an approximate two-layered rewrite theory $\mathfrak{R}(\mathcal{L}^\mathfrak{s})$ that extends a given approximate two-layered rewrite theory $\mathcal{R}(\mathcal{L})$. A first attempt is to define $\mathfrak{R}(\mathcal{L}^\mathfrak{s}) = (\Sigma^\mathfrak{s}, E \cup A, 1R^\mathfrak{s} \cup 2R^\mathfrak{s}, \varepsilon^\mathfrak{s})$ from $\mathcal{R}(\mathcal{L})$ in the same way $\mathcal{L}^\mathfrak{s}$ is obtained from $\mathcal{L}$, but this is not enough to have a coverage-like result. The program `log` in Figure 5 is deterministic and terminating for each $\vartheta(A) \in Int$. So we may execute any instance of it with an approximate encoding $\mathcal{R}$ having no second-layer rules, i.e., $2R = \emptyset$. If $2R^\mathfrak{s} = \emptyset$, then $1R^\mathfrak{s}$ is non terminating because there is an infinite execution corresponding to the case when the value of the program variable `X` in the current configuration is always greater than zero. Another problem is to specify how the strategy ε is extended to $\varepsilon^\mathfrak{s}$. Since it is hard to give general definitions for these questions, we opted for a particular solution that can be implemented in Maude.

Definition 5 (symbolic approximate encoding). *Let $\mathcal{L}^{\mathfrak{s}} = (\Sigma^{\mathfrak{s}}, \mathcal{T}^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$ be the symbolic extension of $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ and $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, 1R \cup 2R, \varepsilon)$ an approximate encoding of $\mathcal{L}$. We assume that there is a total order relation $\prec$ over $1R$ such that:*

1. *the rewrite $t \to^{!}_{(E \cup 1R)/A} \varepsilon(t)$ uses the minimal rule from $1R$ w.r.t. $\prec$ whenever such a rule is applicable;*
2. *if α is unconditional and α' is conditional then $\alpha \prec \alpha'$.*

We let the approximated encoding of $\mathcal{L}^{\mathfrak{s}}$ be $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}}) = (\Sigma^{\mathfrak{s}}, E \cup A, 1R^{\mathfrak{s}} \cup 2R^{\mathfrak{s}}, \varepsilon^{\mathfrak{s}})$:

- *$1R^{\mathfrak{s}} = \{\alpha^{\mathfrak{s}} \mid \alpha \in 1R, \alpha$ unconditional$\}$;*
- *$2R^{\mathfrak{s}} = \{\alpha^{\mathfrak{s}} \mid \alpha \in 1R, \alpha$ conditional$\} \cup \{\alpha^{\mathfrak{s}} \mid \alpha \in 2R\}$;*
- *$\alpha^{\mathfrak{s}} \prec^{\mathfrak{s}} \alpha'^{\mathfrak{s}}$ iff $\alpha \prec \alpha'$;*
- *$\varepsilon^{\mathfrak{s}}$ uses the minimal rule from $1R^{\mathfrak{s}}$ w.r.t. $\prec^{\mathfrak{s}}$.*

Theorem 5 (coverage for approximate rewrite theories). *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition and $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, 1R \cup 2R, \varepsilon)$ be an approximate encoding of $\mathcal{L}$. For every concrete execution $\gamma_0 \to_{\mathfrak{R}(\mathcal{L})} \gamma_1 \to_{\mathfrak{R}(\mathcal{L})} \cdots \to_{\mathfrak{R}(\mathcal{L})} \gamma_n \to_{\mathfrak{R}(\mathcal{L})} \cdots$ there is a symbolic execution $\pi_0 \wedge \phi_0 \to^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_1 \wedge \phi_1 \to^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \to^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_n \wedge \phi_n \to^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$ such that $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ for $i = 0, 1, \ldots$.*

However, the precision relationship between $\mathfrak{R}(\mathcal{L})$ and $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ does not hold in general. The reason is that $1R^{\mathfrak{s}}$ has fewer rules than $1R$ and hence the representative-selection strategy $\varepsilon^{\mathfrak{s}}$ is weaker than ε. Therefore there are no guarantees that the concrete execution given by Corollary 3 will be the same with that chosen by the strategy ε. If the strategy $\varepsilon^{\mathfrak{s}}$ is the "isomorphic image" of ε via the transformation $\bullet \mapsto \bullet^{\mathfrak{s}}$, then the precision result holds:

Theorem 6 (precision for approximate rewrite theories). *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition and $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, 1R \cup 2R, \varepsilon)$ be an approximated encoding of $\mathcal{L}$ such that $1R$ includes only unconditional rules (hence $1R^{\mathfrak{s}} = \{\alpha^{\mathfrak{s}} \mid \alpha \in 1R\}$). For every feasible symbolic execution $\pi_0 \wedge \phi_0 \to_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_1 \wedge \phi_1 \to_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \to_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_n \wedge \phi_n \to_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$ there is a concrete one $\gamma_0 \to_{\mathfrak{R}(\mathcal{L})} \gamma_1 \to_{\mathfrak{R}(\mathcal{L})} \cdots \to_{\mathfrak{R}(\mathcal{L})} \gamma_n \to_{\mathfrak{R}(\mathcal{L})} \cdots$ such that $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ for $i = 0, 1, \ldots$.*

5 Implementing the $\mathbb{K}$ Framework in Maude

The current implementation of the $\mathbb{K}$ framework uses Maude as a rewrite engine. In [4], the framework, at that time called K-Maude, was presented as an extension of Maude consisting in several meta-transformations which gradually translate $\mathbb{K}$ modules into executable Maude modules. In the current version of $\mathbb{K}$ we use a compiler for language definitions where each of these meta-transformations is actually a separate compilation step. Through compilation, $\mathbb{K}$ definitions are translated into Maude rewrite theories which are then used for running/analysing programs. The main components of a $\mathbb{K}$ definition are the syntax declarations, the configuration and the $\mathbb{K}$ (rewrite) rules. To these, the tool adds automatically the rules generated from strictness annotations (e.g. heating/cooling rules 1-4).

This entire work is concerned with how the set of rules is compiled into a two-layered rewrite theory, which is then encoded into Maude by using equations for the first-layer rules and rewrite rules for the second-layer rules. By default, all $\mathbb{K}$ rules are translated into (conditional) equations (i.e. $1R = \mathcal{S}$ and $2R = \emptyset$). This behavior can be altered by specifying (at compile time) that certain rules are to be considered *transitions*, which will trigger their transformation into (conditional) rewrite rules in the resulting Maude module. When using $\mathbb{K}$, one must pass the rule name as an argument for the `-transition` option at compilation time:

```
$ kompile cink.k -transition "division"
```

The above command specifies the rule *division* as a transition; thus, the rule for division is included in $2R$. By this command we express our intent that the tool considers the rule for division as a transition when exploring an execution's transition system. By making it a rewrite rule in Maude, we can explore the nondeterminism generated by the rule when using Maude's *search* command.

Another source of non-determinism arises from strictness annotations. When the *strict* attribute is given to some syntactical construct, the tool chooses by default an arbitrary, fixed order to evaluate its arguments. This optimisation has the side effect of possibly losing behaviours due to missed interleavings. Some of these missed interleavings can be restored using the `-superheat` option. This option is used to instruct the $\mathbb{K}$ tool to exhaustively explore all the non-deterministic evaluation choices for the strictness of a language construct.

Once we know which rules are transitions and which are not, we can easily deduce the two sets $1R$ and $2R$, and thus we obtain the executable rewrite theory $\mathfrak{R}(\mathcal{L})$ as discussed in Section 4. The following example shows how one can explore more behaviours by specifying second-layer rules at compile time. If we compile the language definition of CinK without any options, then running the program `counter` (Figure 5) will result in a single solution, where the return value is either 1 (when the tool first evaluates `dec()` and then `inc()`) or 3 (when it evaluates `inc()` before `dec()`). However, if we set the operation *plus* as superheat:

```
$ kompile cink -superheat "plus"
```

then we obtain both solutions, because the heating rule for addition can be applied in two ways and the option tells the tool to explore them both.

The symbolic transformations discussed in Section 3.2 are implemented as compilation steps in the $\mathbb{K}$ compiler [2]. The tool uses the same translation to Maude discussed above in order to obtain the rewrite theory $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$. An important step in this process is that conditional rules whose conditions cannot be reduced to *true* are compiled as transitions, that is, they are included in $2R$. When performing search in Maude, these rules are essential in exploring all the execution paths, thereby ensuring the Coverage (Theorem 5) property. Note that none of the symbolic transformations applied by the tool to the language definition changes the initial semantics of the language.

The implementation uses a slightly modified version of Maude which includes a hook to the Z3 SMT solver [5] and a corresponding operation called *checkSat*.

It receives as argument an SMTLib string, which is sent to the solver to check its satisfiability. The result returned by the solver is propagated back through the hook to Maude as a string, so *checkSat* can return "sat", "unsat", or "unknown". In practice, our tool uses checkSat to reduce the search space by slicing unfeasible execution paths, which is very important in preserving the precision property. To obtain $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ from a $\mathbb{K}$ definition one uses the symbolic backend as follows:

```
$ kompile cink -backend symbolic
```

This command applies the symbolic transformations, moves the appropriate rules in *2R*, and generates the rewrite theory $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$. Using $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ one can execute programs using either concrete or symbolic values. However, running programs with symbolic values may lead to infinite loops when the loop conditions contain symbolic values. In such cases one can bound the number of execution paths:

```
$ krun log.imp -search -bound 3 -cIN=".List" -cPC="true"
```

This executes `log` (Figure 5) symbolically, until a number of 3 solutions is found. Each solution consists in a result configuration and a formula which constitutes the path condition. The symbolic values are represented as fresh variables with a specific sort (e.g. `A:Int`). These can also be passed as input at the command line of the tool as arguments of the `-cIN` parameter. Users can also set the initial path condition using the `-cPC` option. During the symbolic execution the tool applies a rule only if the next state is feasible: the current path condition and the new conditions imposed by the application of the rule are not "unsat".

6 Conclusion and Related Work

We presented some results that relate language definitions to different kinds of rewrite theories, which encode the language definitions both faithfully and approximately. The results show how (symbolic) analyses performed on a rewrite theory are reflected on the corresponding language definition. The general results are applied to the current implementation of $\mathbb{K}$ language definitions in Maude.

The faithful encoding of $\mathbb{K}$ language definitions as rewrite theories is relatively simple but the resulting theory is not efficient in practice. Therefore we extended the notion of rewrite theory in order to work with under-approximations of the language definitions (and implicitly of the rewrite theories). The approximating theories are more efficient and flexible – the user has the freedom to work with various levels of approximations –, but their use for program analysis must be done with care because they do not preserve all the behavioural properties. The coverage/precision results proved in this paper can help the user in correctly assessing which analyses hold on which representations.

Related Work The first tool supporting $\mathbb{K}$ [4] was written in Maude's meta-level, as a series of transformations translating $\mathbb{K}$ definitions into Maude programs. Then, the $\mathbb{K}$ compiler became a more complex tool that translates a $\mathbb{K}$ definition into an intermediate language, which is used to generate code for various backends, including Maude. The tool and the semantics of $\mathbb{K}$ definitions is described in [8]. The programming-language definition framework presented in this paper (Section 3) is a specialised case of that definition.

The coverage and precision properties, which relate the faithful rewrite-theory encoding of a language and of that language's symbolic version, are analogous to the soundness and completeness results in [10], which relate usual rewriting and rewriting modulo SMT. An interesting alternative to defining symbolic execution as executions in a transformed language (as we do it in [2]) would be to compile a language into a rewriting modulo SMT Maude module.

Our construction of two-layered rewrite theories has some similarities with equational abstractions [9] and with the state-space reduction techniques obtained by transforming rules into equations presented in [6]. However, our first-layer rewrite rules do not equate states as Maude equations do; their semantics is that of transformation, not of equality. Therefore these rules do not have to satisfy the executability and property-preservation requirements of [9,6].

References

1. Standard for Programming Language C++. Working Draft. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.
2. A. Arusoaie, D. Lucanu, and V. Rusu. A generic framework for symbolic execution. In *6th International Conference on Software Language Engineering*, volume 8225 of *LNCS*, pages 281–301. Springer Verlag, 2013. Also available as a technical report at `http://hal.inria.fr/hal-00766220/`.
3. M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.
4. T. F. Şerbanuţă and G. Roşu. K-Maude: A rewriting based tool for semantics of programming languages. In P. C. Ölveczky, editor, *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010*, volume 6381 of *Lecture Notes in Computer Science*, pages 104–122, 2010.
5. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
6. A. Farzan and J. Meseguer. State space reduction of rewrite theories using invisible transitions. In *P. Proceedings of the 21st German Annual Conference on Artificial Intelligence*, pages 142–157. Springer, 2006.
7. D. Lucanu and T. F. Serbanuta. CinK - an exercise on how to think in K. Technical Report TR 12-03, Version 2, Alexandru Ioan Cuza University, Faculty of Computer Science, December 2013.
8. D. Lucanu, T. F. Şerbănuţă, and G. Roşu. The K Framework distilled. In *9th International Workshop on Rewriting Logic and its Applications*, volume 7571 of *Lecture Notes in Computer Science*, pages 31–53. Springer, 2012. Invited talk.
9. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theor. Comput. Sci.*, 403(2-3):239–264, 2008.
10. C. Rocha, J. Meseguer, and C. A. Munoz. Rewriting modulo SMT. Technical Report 218033, NASA, 2013.
11. G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
12. G. Roşu and A. Ştefănescu. Checking reachability using matching logic. In G. T. Leavens and M. B. Dwyer, editors, *OOPSLA*, pages 555–574. ACM, 2012.
13. P. Viry. Equational rules for rewriting logic. *Theor. Comput. Sci.*, 285(2):487–517, 2002.

Towards a Formal Semantics-Based Technique for Interprocedural Slicing*

Irina Măriuca Asăvoae[1], Mihail Asăvoae[1], Adrián Riesco[2]

[1] VERIMAG/UJF, France
{mariuca.asavoae, mihail.asavoae}@imag.fr
[2] Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es

Abstract. Interprocedural slicing is a technique applied on programs with procedures and which relies on how the information is passed at procedure call/return sites. Such a technique computes program slices (i.e. program fragments restricted w.r.t. a given criterion). The existing approaches to interprocedural slicing exploit the particularities of the underlying language semantics in order to compute program slices. In this paper we propose a generic technique for the problem interprocedural slicing. More specific, our approach takes as input a language semantics (given as a rewriting-logic specification) and infers some particularities of it which are further used to compute the program slices.

Keywords: slicing, semantics, Maude, debugging

1 Introduction

Complex software systems are built in a modular fashion where modularity is implemented with functions and procedures in imperative languages, with classes and interfaces in object-oriented programming, with modules in declarative-style programming, or other means of organizing the code. Besides its structural characteristic, the modularity also carries semantic information. For example, the modules could be parameterized by types and values (e.g. the generic classes of Java and C#, the template classes of C++, the parameterized modules of Maude and Ocaml) or could have specialized usability (e.g. abstract classes in object-oriented languages or functors in functional programming languages).

It is preferable, for efficiency and accuracy reasons, that the modular characteristics of a system are preserved during its development and are used for its analysis. As such, it is advisable to integrate the development and the analysis of a system. One possible solution for this integration is to use a formal executable framework such as rewriting logic [10]. The integration methodology proposed by rewriting logic starts with a formal executable semantics of the programming language used for the system development. The formal executable semantics provides the set of all *concrete executions*, for any program correctly

* Research supported by MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

constructed w.r.t. the language syntax and for all possible input data. Next, the notion of a concrete execution could be extended to an *abstract execution*—seen as a program execution with an analysis tool. This is due to the fact that the abstractions and, implicitly, abstract executions are representations of sets of concrete program executions. One particular abstraction is the program slicing [21], which computes safe program fragments (also called slices) w.r.t. a specified set of variables. A complex variant of program slicing, called interprocedural slicing, preserves the modularity of the underlying program and exploits how the program data is passed between the program modules.

Program slicing [21] computes all the program statements that might affect the value of a variable v at a program point of interest, p. The variable v (or a set of variables) is called slicing criterion. The slicing result is standardly computed by incrementally discovering the data dependencies and adding to the slice the statements containing variables dependent of some element in the slicing criterion. The standardized approaches to slicing consider a particular programming language and work around its particularities. We aim to generalize these approaches by giving the slicing method in a generic fashion, using as premises the programming language semantics specification. Namely, we *infer* specific particularities of the language from the semantics specification of the language and use these particularities for the analysis method. For example, in our previous work from [14] we discover *side-effect language constructs* (i.e., constructs triggering data-store changes) which we further use for intraprocedural program slicing. In the current work we make a step forward and describe an interprocedural slicing method which we use to identify which are the slicing prerequisites, i.e., the semantics entities that we need to infer in order to claim generality of the slicing w.r.t. the language semantics. We identify these prerequisites as being *context-updates constructs* and *parameter passing patterns*, besides the already settled side-effect constructs from [14]. Next, we explain the terminology used.

Interprocedural slicing is the slicing method applied on programs with procedures where the slice is computed for the entire program by taking into account the procedure calls/returns. The main problem that arises in interprocedural slicing, comparing to e.g., intraprocedural slicing, is related to the fact that the procedure calls/returns may be analyzed with a too coarse abstraction. Usually, the coarse abstraction relies only on the call graph without taking into account the context changes occurring during a procedure call/return. By *context changes* we refer to the instantiation of the local variables during and after a procedure execution and we name the procedure call/return language constructs as *context-update constructs*. By *parameter passing pattern* we understand the manner in which some variables get to traverse the context-updates.

In this paper we extend our previous work on generic slicing from [14] from intraprocedural to interprocedural generic slicing. Hence, we consider given a language semantics $\mathcal{S}$, and we use intraproceudral slicing to construct the interprocedural method of slicing $\mathcal{S}$-programs (i.e., programs written in the language specified by $\mathcal{S}$, i.e., well-formed terms in $\mathcal{S}$). We consider $\mathcal{S}$ to be a rewriting logic theory [10] which is executable and benefits of tool support via the Maude

system [2], an implementation of the rewriting logic framework. We use this study of the interprocedural program slicing to identify the which are the prerequisites of a generic slicing. Namely, along studying the interprocedural slicing we identify these prerequisites as being the context-updates together with parameter passing. This technique is concretized with an implementation into a generic semantics-based slicing tool developed in Maude, but for the moment the prerequisites are given manually. Hereafter, we give more details about the settings of our interprocedural $\mathcal{S}$-program slicing.

Firstly, we consider $\mathcal{S}$ to be from the family of imperative and object-oriented languages. Many constructs in these languages use the notion of *scope* which delimits pieces of program, e.g., the loops or function bodies. These types of scoping information are explicitly represented and manipulated in the rewriting logic definitions of C [3] or Java [4]. We use here, as a case study for $\mathcal{S}$, an extension of the WHILE language [7] in which we introduce variable scoping (i.e., homonymous variables behave differently w.r.t. the current scope). We call this extension WhileF which is a simple imperative language with conditions and loops, enriched with constructs like functions and local variables.

Secondly, the program slicing step of our technique receives as input both context-update and side-effect information hence the relatively simple slicing step in [14] is subject to heavy changes. For one, it is necessary to address the representability of the derived context-update constructs w.r.t. the interprocedural program slicing. Namely, a combined representation of context-update and side-effect constructs could be terms denoting generic skeletons for *procedure summaries*—a succinct representation of the procedure behavior w.r.t. its input variables, as introduced in [17] and advanced in [5]. Our approach is complementary to it, because it creates language-specific representations of summaries, based on the specific manipulation of context-updates in the formal semantics.

Finally, the interprocedural slicing proposed here has the following status w.r.t. the Maude implementation: Our work in [14] containing the meta-analysis for side-effect constructs and the intraprocedural slicing is already prototyped in Maude. For the current work, we defined WhileF (see the source at `http://maude.sip.ucm.es/slicing/`) and we set the interprocedural slicing to work with the context-updates, side-effects, and intraprocedural slicing. However, the meta-analysis producing context-updates is, at the moment, at conceptual level, being used as an hand-given input. It also remains to implement the procedure summaries which improve the accuracy of the slicing result as well as the procedure parameter passing pattern extraction.

Related work. Program slicing is a program analysis technique which computes safe program slices (i.e., sequences of program statements) w.r.t. a given criteria (i.e., sets of variables of interest). Program slicing addresses a wide range of applications from code parallelization [19] to program testing [6], debugging [18], and static analysis [9,8]. We note first that our work resides at the intersection of standard program analysis and rewriting logic. As such, we organize the related work presentation in two categories: *(A) standard slicing* as defined in the already classical program analysis and *(B) rewriting logic* from the perspective

of its preexistent *analysis tools* and its programming languages *semantics specifications*. We use this section to better localize our work rather than to give a direct comparison with other researches. The reasons for this setting are: (1) even though program slicing is relatively well established in program analysis, in rewriting logic is rather a new topic; (2) other work on slicing in rewriting logic apply on model checking counterexamples rather than programs while other program analysis tools for language semantics specifications in rewriting logic target other topics, e.g., testing and debugging.

(A) Program slicing was introduced in [21] where, for a given program with procedures, a limited form of context information (i.e., procedure call location) is used to compute the program slices. The approach resembles an on-demand procedure inlining, using a backward propagation mechanism. The results of [21] are backward program slices. Moreover, multiple procedure calls are included in the computed slice without distinguishing between them w.r.t. their intra-procedural information. Our proposed approach takes into consideration the context-update constructs (as extracted from the formal semantics) and produces forward slices (via term slicing on the program term). Moreover, the context-update constructs induce the symbolic procedure summaries as in [17,8,5]. A procedure summary is a compact representation of the procedure behavior, parameterized by its input values and, in our proposed framework, is derived from the context-update constructs and intraprocedural slicing. The interprocedural slicing is explicit in [8] and implicit in [17,5] and sets the support for interprocedural program analyses. Next we compare our work with the underlying interprocedural slicing algorithm of each of the three aforementioned approaches.

The work in [17] uses a data-flow analysis to represent how the information is passed between procedure calls. It is applied on a restricted class of programs—restricted by a finite lattice of data values—, while the underlying program representation is a mix of control-flow and call graphs. In comparison, our approach considers richer context information (as in [8]) while working on a term representation of a program. Hence, we have an implicit representation of the control-flow and call graphs. The work in [5] keeps the same working structures but addresses the main data limitation of [17]. As such, the procedure summaries are represented as sets of constraints on the input/output variables. The underlying interprocedural slicing algorithm of [5] is more refined than our approach just because of the sharper representation of context information. Our approach requires a specialized data abstraction on top of the interprocedural slicing procedure—this is one of our proposed developments. We follow closely the work in [8] which introduces a new program representation for interprocedural slicing. This is a mix program graph with explicit representation (as specialized nodes and edges) of context information and the interprocedural slicing is computed on this program representation. In comparison, our approach does not require the explicit context representation but uses term matching (in the second part of the algorithm) to distinguish between different context. We present [8] more details in Section 2. What separates our approach from the three aforementioned techniques is that we work with a formal executable semantics, in

the rewriting logic framework. This means, on one hand, that the results of our slicing algorithm are correct w.r.t. the language specification and, on the another hand, that the slicing algorithm is generic (as the slicing information is extracted by meta-processing the formal semantics).

(B) In the rewriting logic environment there are several existing approaches towards program debugging, testing, and analysis. The work in [12] presents an approach to generate test cases similar to the one presented here namely, both use the formal specifications of the languages semantics to extract information about programs written in these languages. In this case, the semantic rules are used to instantiate the state of the variables used by the given program by using narrowing. In this way, it is possible to compute the values of the variables required to traverse all the statements in the program, the so called coverage. Similarly, in [13] is presented an approach to perform declarative debugging of programming languages defined by either their big-step or small-step semantics. It describes a generic way to build the proof trees for the execution of programs in such a way that, given some information from the user, it can locate the function responsible for the error.

The recent work in [1] proposes a first slicing technique of rewriting logic computations. It takes as input an execution trace—the result of executing Maude model checker tools—and computes dependency relations using a backward tracing mechanism. While they perform dynamic slicing by executing the semantics for an initial given state, we propose a static approach that is centered around the rewriting logic theory of the language specification. Moreover, our main target application is not counterexamples or execution traces of model checkers, but programs executed by the particular semantics.

The proposed technique follows our previous work on language-independent program slicing in rewriting logic environment [14]. Actually, the implementation of the current work is an extension of the slicing tool from [14]. Both approaches share the methodology steps: (1) the initial meta-analysis of $\mathcal{S}$ and (2) the program analysis conducted over the $\mathcal{S}$-programs. Namely, in [14] we use the classical WHILE language augmented with side-effect constructs (assignments and read/write statements) to exemplify (1) the inference of the set of side-effect language constructs in $\mathcal{S}$ and (2) the program slicing as term rewriting.

As a semantical framework, Maude [2] and rewriting logic [10] have been used to specify the semantics of several languages, such as LOTOS [20], CCS [20], Java [4], or C [3]. These researches, as well as several other efforts to describe a methodology to represent the semantics of programming languages in Maude, led to the *rewriting logic semantics project* [11]—which presents a comprehensive compilation of these works—and to the development of $\mathbb{K}$ [16]—a tool built upon a continuation-based technique that provides mechanisms to (i) ease language definitions and (ii) translate these definitions into Maude which comes with its "for free" analysis tools—. Our interest in these semantics consists in using them as benchmarks to validate different meta-analyses as, e.g., the context-update inference. First, we plan to limit our approach to those semantics directly

implemented in Maude, but we would like to further extend our attention to the semantics implemented in $\mathbb{K}$ and automatically interpreted in Maude.

The rest of the paper is organized as follows: Section 2 introduces some foundations of slicing, while Section 3 describes our proposed interprocedural slicing algorithm pivoting on an example. In Section 4 we conclude and present some lines for future work.

2 Preliminaries

Program slicing, as introduced in [21], is a program analysis technique which computes all the program statements that might affect the value of a variable v at a program point of interest, p. It is a common setting to consider p as the last instruction of a procedure or the entire program. Hence, without restricting the proposed methodology, here we consider slices of the entire program.

A classification of program slicing techniques identifies *intraprocedural* slicing when the method is applied on a procedure body and *interprocedural* slicing when the method is applied across procedure boundaries. The key element of a methodology for interprocedural slicing is the notion of context (i.e. the values of the function/procedure parameters). Next, we elaborate on how context-aware program slicing produces better program slices than a context-forgetful one.

Let us consider, in Fig. 1, the program from [8], written as an WhileF program term, upon which we present subtleties of interprocedural slicing. We start the slicing with the set of variables of interest {`z`}.

The first method, in [21], resembles an on-demand inlining of the necessary procedures. In the example in Fig. 1, the variable {`z`} is an argument of procedure `Add` call in `Inc`, hence, the sliced body of `Add` is included in the slice of `Inc`. Note that, when slicing the body of `Add`, `z` is replaced by `a` hence, the slicing of `Add` deems {`a`} and {`b`} as relevant. The return statement of procedure `Inc` is paired with the call to `Inc`, in the body of `A` so the variable {`y`} becomes relevant for the computed slice. When the algorithm traces the source of the variable `y`, it finds the second call to `Add` in the body of `A` (with the arguments `x` and `y`) and includes it in the program slice. When tracing the source of `x` and `y`, it leads to include the entire body of procedure `Main` (through the variables `sum` and `i`, which are used by the assignments and calls of `Main`). Using this method, the program slice w.r.t. the set of variables of interest - {`z`}, is the original program, as in Fig. 1. This particular slice is a safe over-approximation of a more precise one (which we present next) because the method relies on a transitive-closure—fixpoint computation style where all the variables of interest are collected at the level of each procedure body. As such, the body of procedure `Add` is included twice in the computed slice.

The second approach in [8] exploits, for each procedure call, the available information w.r.t. the program variables passed as arguments (i.e., the existing context before the procedure call). Again, in the example in Fig. 1, the variable `z` is an argument of procedure `Add` hence, upon the return of `Add`, its body is included in the slice. However, because of the data dependencies between

```
function Main (){
 sum := 0;
 Local i;
 i := 1;
 while i < 11 do
  call A (sum, i)
}
```

```
function A (x, y) {
 call Add (x, y);
 call Inc (y)
}
function Add (a, b) {
 a := a + b
}
```

```
function Inc (z) {
 Local i;
 i := 1;
 call Add (z, i)
}
```

Fig. 1. A WhileF program `Px` with procedures `Main`, `A`, `Add`, and `Inc`.

variables `a` and `b` (with `a` using an unmodified value of `b`) only the variable `a` is collected and further used in slicing. Next, upon the return statements of `Add` and then `Inc`, the call of `Inc` in `A` (with parameter `y`) is included in the slice. Note that the call to `Add` from `A` (with parameters `x` and `y`) is not included in the slice because it does not modify the context (i.e., the variables of interest at the call point in `A`). As such, the slicing algorithm collects only the second parameter of procedure `A`, and following the call to `A` in `Main`, it discovers `i` as the variable of interest (and not `sum` as it was the case of the previous method). Hence, the sliced `A` with only the second argument is included in the computed slice. Consequently, the variable `sum` from `Main` is left outside the slice. The result is presented in Fig. 2.

```
function Main (){
 Local i;
 i := 1;
 while i < 11 do
  call A (i)
}
```

```
function A (y) {
 call Inc (y)
}
function Add (a, b) {
 a := a + b
}
```

```
function Inc (z) {
 Local i;
 i := 1;
 call Add (z, i)
}
```

Fig. 2. The result of a context-dependent interprocedural analysis for `Px`.

Any analysis that computes an interprocedural slice works with the control-flow graph—which captures the program flow at the level of procedures— and the call graph—which represents the program flow between the different procedures—. To improve the precision of the computed program slice, it is necessary for the analysis to use explicit representations of procedure contexts (as special nodes and transitions). This is the case of the second method which relies on a program representation called *system dependence graph*.

3 Semantics-based Interprocedural Slicing

We present in this section the algorithm for our interprocedural slicing approach, and illustrate it with an example. Then, we describe a Maude prototype executing the algorithm for semantics specified in Maude.

3.1 Program slicing as term slicing

In [14] we show how to extract the set of side-effect language instructions SE from the semantics specification $\mathcal{S}$ and how to use SE for an *intraprocedural slicing* method. In the current work we focus on describing the *interprocedural slicing method* which is built on top the intraprocedural slicing result from [14].

The programs written in the programming language specified by $\mathcal{S}$ are denoted as $\mathfrak{p}$. By *program variables* we understand subterms of $\mathfrak{p}$ of sort *Var*. If we consider the *subterm* relation as $\preccurlyeq$, we have $v \preccurlyeq \mathfrak{p}$ where v is a program variable.

We consider a *slicing criterion sc* to be a subset of program variables which are of interest for the slice. We denote by SC the slicing criterion sc augmented with *data flow information* that is collected along the slicing method. Hence, SC is a set of pairs of program variables of form $v, \overset{\curvearrowleft}{v'}$, denoting that v depends on v', or just variables v, denoting that v is independent.

We assume as given the set of program functions $\mathfrak{F}_\mathfrak{p}$ defining the program $\mathfrak{p}$. We claim that $\mathfrak{F}_\mathfrak{p}$ can be inferred from the term $\mathfrak{p}$, given the $\mathcal{S}$-sorts defining functions, variables, and instruction sequences. We base this claim on the fact that $\mathfrak{p}$ is formed, in general, as a sequence of function definitions hence its sequence constructor can be automatically identified from $\mathcal{S}$. Also, we denote by $\text{getFnBody}(f, \mathfrak{F}_\mathfrak{p})$ the matching of the body of function f in the set $\mathfrak{F}_\mathfrak{p}$. Note that $\text{getFnBody}(f, \mathfrak{F}_\mathfrak{p}) \preccurlyeq \mathfrak{F}_\mathfrak{p}$.

Furthermore, we denote the method computing the intraprocedural slicing as $\$(B, SC, SE)$, where B is the code, i.e., the body of some function f in $\mathfrak{p}$. Obviously, we have $B \preccurlyeq \mathfrak{p}$. The result of $\$(B, SC, SE)$ is a term $SC :: fn\langle fn(fp^\sharp)\{fs\}\rangle$ where SC is the data flow augmented slicing criterion, $fn \in \mathit{FunctionName}$ is a function identifier, and fs is the slice computed for fn. Meanwhile $fp^\sharp$ is the list of fn's formal parameters fp filtered by SC, i.e., all the formal parameters not appearing in SC are abstracted to a fixed additional variable $\sharp$.

Now we give a brief explanation on how the intraprocedural slicing $ works. We say that a program subterm *modifies* a variable v if the top operator is in SE and v appears as a leaf in a specific part of the subterm (e.g., the variable v appears in the first argument of `_:=_`, or in `Local_`). When such a subterm is discovered by $ for a slicing variable then the slicing criterion is updated by adding the variables producing the side-effects (e.g., all variables v' in the second argument of `_:=_`) and the data flow relations $v, \overset{\curvearrowleft}{v'}$. We call fs a *skeleton subterm* of B and we denote this as $fs \precsim B$.

In Fig. 3 we give the slicing method, **termSlicing**, which receives as input the slicing criterion sc, the set of program functions $\mathfrak{F}_\mathfrak{p}$, and the set of side-effect and context-updates syntactic constructs, SE and CU, respectively. The output is the set of sliced function definitions *slicedFnSet* together with the obtained data flow augmented slicing criterion *dfsc*. Note that $\mathfrak{F}_\mathfrak{p}$, SE, and CU are assumed to be precomputed based on the programming language semantics specification $\mathcal{S}$. The algorithm for inferring SE is given in [14, Section 4]. The algorithm for inferring CU goes along the same lines as the one for SE and it is based on the automatic discovery of stack structures used in $\mathcal{S}$ for defining the

programming language commands. For example, in WhileF the only command inducing context-updates is `Call_(_)` instruction. In the current work we assume CU given in order to focus on the interprocedural slicing as term slicing method. However, we claim that **termSlicing** is generic w.r.t. $\mathcal{S}$ since $\mathfrak{F}_{\mathtt{p}}$, SE, and CU can be automatically derived from $\mathcal{S}$.

termSlicing is a fixpoint iteration which applies the *current* data-flow-augmented slicing criterion over the function terms in order to discover new skeleton subterms of the program that comply with the slicing criterion. The protocol of each iteration step is to take each currently sliced function and slice down and up in the *call graph*. In other words, the *intraprocedural slicing* is applied on every *called function* (i.e., goes *down* in the call graph) and every *calling function* (i.e., goes *up* in the call graph).

Technically, **termSlicing** relies on incrementally building the program slice in the *workingSet* variable and the data flow augmented slicing criterion, *dfsc*. This process has two phases: the initialization of the *workingSet* and *dfsc* (lines 0-6) and the loop implementing the fixpoint (lines 7-39).

The initialization part computes the slicing seed for the fixpoint by independently applying the intraprocedural slicing $\$(\_,\_,\_)$ with the slicing criterion sc for each function in the program $\mathtt{p}$. The notation $A \cup\!= B$ (line 3) stands for "A becomes $A \cup B$" where $\cup$ is the set union. Similarly, $A \uplus\!= B$ (line 4) is the union of two data dependency graphs. Namely, $A \uplus B$ is the set union for graph edges filtered by the criterion that if a variable v is independent in A but dependent in B (i.e., there exists an edge $\_ \curvearrowright \_$ with v on one of the ends) then the independent variable v is eliminated from $A \uplus B$. For example, the initialization step applied on the program in Fig. 1 produces the following *workingSet*$'$: $\mathtt{z} :: \mathtt{Inc}\langle \mathtt{Inc(z)}\{\mathtt{Call\ Add(z,\sharp)}\}\rangle, \emptyset :: \mathtt{Main}\langle \mathtt{Main()}\{\}\rangle, \emptyset :: \mathtt{A}\langle \mathtt{A(\sharp,\sharp)}\{\}\rangle, \emptyset :: \mathtt{B}\langle \mathtt{B(\sharp,\sharp)}\{\}\rangle, \emptyset :: \mathtt{Add}\langle \mathtt{Add(\sharp,\sharp)}\{\}\rangle$.

The fixpoint-loop (lines 7-39) discovers the call graph in an on-demand fashion using the context-update set CU which directs the fixpoint-iteration towards applying the slicing on the called/calling function. As such, when a context-update (i.e., `Call_(_)` in the semantics of WhileF) is encountered in the current slice, we proceed to slice *the called function* (lines 10-19). Next, when a context-update of the currently considered functions is encountered, we proceed again to slice *the calling function* (lines 20-29). Each time we update the current data-flow-augmented slicing criterion and the slice of the current function (lines 30-36). We iterate this process until the skeleton subterm of every function is reached, i.e., *workingSet* is stable. Note that the stability of *workingSet* induces the stability of *dfsc*, the data flow augmented slicing criterion.

We now describe in more details each of the three parts of the fixpoint-loop: the *called* (lines 10-19), the *calling* (lines 20-29), and the *current* (lines 30-36) functions. The *called* and *calling* parts have a similar flow with slight differences in the operators used. They can be summarized as:

$$SC \uplus\!= SC\ {}^{fn}\lfloor_{fnCalled}\ \mathit{filtered}\$(\mathit{fnCalled}, \sqsubseteq)\ {}_{fnCalled}\rfloor^{fn}$$

$$SC \uplus\!= SC\ {}_{fn}\lceil^{fnCalling}\ \mathit{filtered}\$(\mathit{fnCalling}, \sqsupseteq)\ {}^{fnCalling}\rceil_{fn}$$

termSlicing
Input: $sc, \mathfrak{F}_\mathfrak{p}, SE, CU$
Output: $slicedFnSet, dfsc$
0 $workingSet' := \emptyset$; $dfsc := \emptyset$;
1 **for all** $fn(args)\{fnBody\} \in \mathfrak{F}_\mathfrak{p}$ **do**
2 $SCinit :: fn\langle fnInitSlice\rangle := \$(fnBody, \{x \in sc \mid x \preccurlyeq fs \text{ or } x \preccurlyeq args\}, SE)$;
3 $workingSet' \cup= \{SCinit :: fn\langle fnInitSlice\rangle\}$;
4 $dfsc \uplus= SCinit$;
5 **od**
6 $workingSet := \emptyset$;
7 **while** $workingSet \neq workingSet'$ **do**
8 $workingSet := workingSet'$;
9 **for all** $SC :: fn\langle fnSlice\rangle \in workingSet$ **do**
10 $wsFnCalled := \emptyset$;
11 **for all** $Call \in CU$ **for all** $Call\ fnCalled \preccurlyeq fnSlice$ **do**
12 $fnCldSC := SC\ ^{fn}\lfloor_{fnCalled}$;
13 **for all** $fnCldSCPrev :: fnCalled\langle\_\rangle \in workingSet$ **do**
14 **if** $fnCldSC \sqsubseteq fnCldSCPrev$ **then break**;
15 $fnCldBd := \text{getFnBody}(fnCalled, \mathfrak{F}_\mathfrak{p})$;
16 $fnCldSCNew :: fnCalled\langle fnCldSlice\rangle := \$(fnCldBd, fnCldSC, SE)$;
17 $wsFnCalled \cup= \{fnCldSCNew :: fnCalled\langle fnCldSlice\rangle\}$;
18 $SC \uplus= fnCldSCNew\ _{fnCalled}\rfloor^{fn}$;
19 **od**
20 $wsFnCalling := \emptyset$;
21 **for all** $Call \in CU$ **for all** $fnCalling \in \mathfrak{F}_\mathfrak{p}$ s.t. $Call\ fn \preccurlyeq fnCalling$ **do**
22 $fnClgSC := SC\ _{fn}\lceil^{fnCalling}$;
23 **for all** $fnCallingSCPrev :: fnCalling\langle\_\rangle \in workingSet$ **do**
24 **if** $fnClgSC \sqsupseteq fnCallingSCPrev$ **then break**;
25 $fnClgBd := \text{getFnBody}(fnCalling, \mathfrak{F}_\mathfrak{p})$;
26 $fnClgSCNew :: fnCalling\langle fnClgSlice\rangle := \$(fnClgBd, fnClgSC, SE)$;
27 $wsFnCalling \cup= \{fnClgSCNew :: fnCalling\langle fnClgSlice\rangle\}$;
28 $SC \uplus= fnClgSCNew\ ^{fnCalling}\rceil_{fn}$;
29 **od**
30 $fnBd := \text{getFnBody}(fn, \mathfrak{F}_\mathfrak{p})$;
31 $SCNew :: fn\langle fnSliceNew\rangle := \$(fnBd, SC, SE)$;
32 $dfsc \uplus= SCNew$;
33 **for all** $Call \in CU$ **for all** $Call\ fnCalled \preccurlyeq fnSliceNew$ **do**
34 **if** $\_ :: fnCalled\langle\_\{\}\rangle \in wSetFnCalled$ **then**
35 $fnSliceNew := \text{erraseSubterm}(Call\ fnCalled, fnSliceNew)$
36 **od**
37 $workingSet' \Cup= \{SCNew :: fn\langle fnSliceNew\rangle\} \Cup wsFnCalled \Cup wsFnCalling$;
38 **od**
39 **od**
40 $slicedFnSet := \text{get}\langle\rangle\text{Content}(workingSet)$

Fig. 3. Program slicing as term slicing algorithm.

where *fn* is the name of the current function, *fnCalled* is the name of a functions called from *fn*, and *fnCalling* is the name of a function which is calling *fn*.

The operators $^{-}\lfloor_{-}$ and $_{-}\rfloor^{-}$ stand for the abstraction of the slicing criterion *downwards in the calling graph* from *fn* into *fnCalled* and back, respectively. The abstraction ${}^{fn}\lfloor_{fnCalled}$ pivots on the actual parameters of *fnCalled* and, based on patterns of function calls, it maps the actual parameters of *fnCalled* from the current environment $SC :: fn$ into the environment of *fnCalled*. The abstraction ${}_{fnCalled}\rfloor^{fn}$ renders the reverse mapping from the (sliced) called environment back into the current one. Similarly for $_{-}\lceil^{-}$ and $^{-}\rceil_{-}$ operators which perform the abstraction *upwards in the call graph* from *fn* to *fnCalling*, pivoting on the parameters of *fn*. For example, for program `Px` from Fig. 1 we have $\overset{\frown}{\texttt{z}},\texttt{i}\ {}^{\texttt{Inv}}\lfloor_{\texttt{Add}}\ \overset{\frown}{\texttt{a}},\texttt{b}\ {}_{\texttt{Add}}\rfloor^{\texttt{Inc}}\ \overset{\frown}{\texttt{z}},\texttt{i}$ and $\overset{\frown}{\texttt{z}},\texttt{i}\ {}_{\texttt{Inc}}\lceil^{\texttt{A}}\ \overset{\frown}{\texttt{y}},\sharp{=}\texttt{y}\ {}^{\texttt{A}}\rceil^{\texttt{Inc}}\ \overset{\frown}{\texttt{z}},\texttt{i}$. For the current work, the only pattern of function calls that we have experimented is the *complete list of call-by-reference parameters*; we plan to experiment with others in the future.

The operator *filtered*(fnC, rel) (lines 13-17 and 23-27) is a *filtered* slicing of fnC, where the filter is a relation between the current abstraction of SC and previously computed slicing criterions for the called/calling function fnC. We say that $SC \sqsubseteq SCPrev$ if SC is a subgraph of $SCPrev$ such that there is no edge $v, \overset{\frown}{v'}$ in $SCPrev$ such that v is a node in SC and v' is a function parameter which is not in SC. This means that SC has no additional dependent data v' among the function parameters that should participate to the current slicing criterion. Meanwhile, $SC \sqsupseteq SCPrev$ is defined as $SCPrev \sqsubseteq SC$ due to the fact that now the sense in the dependency graph is reversed and so the slicing criterion in the calling function ($SCPrev$) is the one to drive the reasoning. Hence, if the filter relation is true then the new slice is not computed anymore (lines 14 and 24) because the current slicing criterion is subsumed in the previous computation.

Finally, in lines 30-36 we compute a new slice for the current function *fn* from which we eliminate any context-update subterm *Call fnCalled* for which the currently computed slice of *fnCalled* has an empty body (lines 33-36). In line 37 we collect all the slices computed at the current iteration in $workingSet'$. Note that $_{-}\Cup_{-}$ operator is an *abstract union* which first computes the equivalence class of slices for each function, based on the graph inclusion of the data-flow-augmented slicing criterion, and then performs the union of the results.

Recall that, in Section 2, we describe two interprocedural slicing methods presented in [21] and [8], being the second more precise than the first one. In our approach the difference is based solely on the data flow relation we use for \$. Hence, we can distinguish two types of **termSlicing**: the *naïve* one where the data flow relations are ignored and the *savvy* one which collects and uses data flow relations. Note that the data flow relation is currently assumed as given but we plan to investigate the automatic inference of the data flow relation from $\mathcal{S}$.

For example, the iterations of the savvy **termSlicing** for the program `PX` in Fig. 4 and the slicing criterion {`z`} are listed in Fig. 5. Namely, in the first boxed rows the slicing criterion {`z`} is applied on $\mathfrak{F}_{\texttt{PX}}$ to produce the skeleton subterms used as the fixpoint seed. Hence, the fixpoint seed contains one nonempty skeleton as `z` appears only in `Inc`. Note that `i`—the second parameter of `Call Add`—is abstracted to $\sharp$ as no data dependency is currently determined for it.

```
Main () {
 sum := 0;
 Local i, j;
 i := 1; j := -1;
 While i < 11 Do
  Call A (sum, i);
  Call B (sum, j);
  Call A (sum, j)
}
```

```
A (x, y) {
 If x > 1 Then
  Call Add(x, y);
  Call Inc (y)
}
B (x, y) {
 If x > 0 Then
  Call B(x + y, y)
}
```

```
Add (a, b) {
 a := a + b
}
Inc (z) {
 Local i, j;
 i := 1; j := i;
 Call Add (z, i);
 Call Inc (j)
}
```

Fig. 4. PX—the extension of the WhileF program Px.

In the second box of rows we consider the slicing criterion for Inc—the only one nonempty from the seed—and we iterate the fixpoint for it. The first row deals with the (only) called function appearing in Inc's skeleton, namely Add(z, ♯). Note that the slicing criterion z is abstracted downwards in the call graph so the slicing criterion becomes a, the first formal parameter of Add. The slice of Add with {a} as slicing criterion is showed in the third column while the slicing criterion becomes $\overset{\curvearrowleft}{\mathtt{a}}, \mathtt{b}$, i.e., a depends on b. Because b is a formal parameter, it gets abstracted back in Inc as Add's actual parameter i. Hence, the updated criterion used in Inc is $\overset{\curvearrowleft}{\mathtt{z}}, \mathtt{i}$ and it is used for the calling function A, in the second row, and also for the recursive call to Inc itself, in the third row. In these rows, the slicing criterion is abstracted upwards in the call graph and the formal parameter z becomes y in A and j in Inc. Meanwhile i is ruled out (becomes ♯) because it is not a parameter and hence it is not relevant in a calling function. The fourth row shows the computation of Inc's skeleton based on the current slicing criterion $\overset{\curvearrowleft}{\mathtt{z}}, \mathtt{i}$. Furthermore, upon performing the abstract union ⩏ at the end of the fixpoint iteration, then Inc's skeleton is Inc(z){...}.

The fixpoint iteration continues in the third box by adding to the slice the function Main due to the upward phase (since Main contains a call to A). The upward parameter substitution of y from A is i in Main and the slice of Main is updated in the third row. Note that the □ in all the other rows signifies the reach of the **break** in lines 14 or 24 in **termSlicing** and stands for "nothing to be done." The fourth box contains the final step of the fixpoint when there is nothing else changed in $workingSet'$ (i.e., all the rows contain □ in the last column). Hence, for the example in Fig. 4 we obtain the slice in Fig. 2 with the only difference that the sliced Inc is now the entire Inc from Fig. 4 (due to the newly added assignment "j:=i").

termSlicing terminates because there exists a finite set of function skeleton subterms, a finite set of data flow graphs, a finite set of edges in the call graph for each function, and any loop in the call graph is solved based on the data flow graph ordering. Moreover, **termSlicing** produces a valid slice because it exhaustively saturates the slicing criterion. However, the obtained slice is not minimal due to the skeletons union ⩏. Still, there is a consistent difference between the

Slicing variables	**Function contexts**	**Computed slice** (identified subterms)
$\mathtt{z} :: \top$	${}^{\top}\lfloor_{\mathtt{Inc}}\ \mathtt{z} \to \mathtt{z}\ {}_{\mathtt{Inc}}\rfloor^{\top}$	$\mathtt{Inc}(\mathtt{z})\ \{\mathtt{Call}\ \mathtt{Add}(\mathtt{z},\sharp)\}$
	${}^{\top}\lfloor_{\mathtt{Main}}\ \sharp = \emptyset\ {}_{\mathtt{Main}}\rfloor^{\top}, \ldots$	$\mathtt{Main}()\{\}, \mathtt{A}(\sharp,\sharp)\{\}, \mathtt{B}(\sharp,\sharp)\{\}, \mathtt{Add}(\sharp,\sharp)\{\}$
$\mathtt{z} :: \mathtt{Inc}$	${}^{\mathtt{Inc}}\lfloor_{\mathtt{Add}}\ \mathtt{a} \to \overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}}\ {}_{\mathtt{Add}}\rfloor^{\mathtt{Inc}}\ \overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}}$	$\mathtt{Add}(\mathtt{a},\mathtt{b})\ \{\mathtt{a} := \mathtt{a} + \mathtt{b}\}$
$\overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc}$	${}_{\mathtt{Inc}}\lceil^{\mathtt{A}}\ \mathtt{y}, \overset{\curvearrowleft}{\sharp} \to \mathtt{y}, \overset{\curvearrowleft}{\sharp}\ {}^{\mathtt{A}}\rceil_{\mathtt{Inc}}\ \overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}}$	$\mathtt{A}(\sharp,\mathtt{y})\ \{\mathtt{Call}\ \mathtt{Inc}(\mathtt{y})\}$
$\overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc}$	${}_{\mathtt{Inc}}\lceil^{\mathtt{Inc}}\ \overset{\curvearrowleft}{\mathtt{j}}, \sharp \to \overset{\curvearrowleft}{\mathtt{j}}, \overset{\curvearrowleft}{\mathtt{i}}\ {}^{\mathtt{Inc}}\rceil_{\mathtt{Inc}}\ \overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}}$	$\mathtt{Inc}(\sharp)\{\mathtt{Local}\ \mathtt{i},\mathtt{j};\ \mathtt{i}{:=}1;\ \mathtt{j}{:=}\mathtt{i};\ \mathtt{Call}\ \mathtt{Add}(\sharp,\mathtt{i});\ \mathtt{Call}\ \mathtt{Inc}(\mathtt{j})\}$
$\overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc}$	$\overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} \to \overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}}$	$\mathtt{Inc}(\mathtt{z})\{\mathtt{Local}\ \mathtt{i};\ \mathtt{i}{:=}1;\ \mathtt{Call}\ \mathtt{Add}(\mathtt{z},\mathtt{i})\}$
$\mathtt{y} :: \mathtt{A}$	${}^{\mathtt{A}}\lfloor_{\mathtt{Add}}\ \mathtt{b} \sqsubseteq \overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} :: \mathtt{Add}\ {}_{\mathtt{Add}}\rfloor^{\mathtt{A}}\ \mathtt{y}$	$\square$
$\mathtt{y} :: \mathtt{A}$	${}^{\mathtt{A}}\lfloor_{\mathtt{Inc}}(\mathtt{z} \sqsubseteq \overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc})\ {}_{\mathtt{Inc}}\rfloor^{\mathtt{A}}\ \mathtt{y}$	$\square$
$\mathtt{y} :: \mathtt{A}$	${}_{\mathtt{A}}\lceil^{\mathtt{Main}}\ \mathtt{i} \to \mathtt{i}\ {}^{\mathtt{Main}}\rceil_{\mathtt{A}}\ \mathtt{y}$	$\mathtt{Main}()\{\mathtt{Local}\ \mathtt{i};\mathtt{i}{:=}1;\mathtt{While}\ \mathtt{i}{<}1\ \mathtt{Do}\ \mathtt{Call}\ \mathtt{A}(\sharp,\mathtt{i})\}$
$\mathtt{y} :: \mathtt{A}$	$\mathtt{y} = \mathtt{y} :: \mathtt{A}$	$\square$
$\overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} :: \mathtt{Add}$	${}_{\mathtt{Add}}\lceil^{\mathtt{Inc}}(\overset{\curvearrowleft}{\mathtt{z}}, \mathtt{i} \sqsupseteq \overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc})\ {}^{\mathtt{Inc}}\rceil_{\mathtt{Add}}\ \mathtt{y}$	$\square$
$\overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} :: \mathtt{Add}$	${}_{\mathtt{Add}}\lceil^{\mathtt{A}}(\overset{\curvearrowleft}{\mathtt{x}}, \overset{\curvearrowleft}{\mathtt{y}} \sqsupseteq \mathtt{y} :: \mathtt{A})^{\mathtt{A}}\rceil_{\mathtt{Add}}\ \mathtt{y}$	$\square$
$\overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} :: \mathtt{Add}$	$\overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} = \overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} :: \mathtt{Add}$	$\square$
$\overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc}$	${}^{\mathtt{Inc}}\lfloor_{\mathtt{Add}}(\overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} \sqsubseteq \overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} :: \mathtt{Add})\ {}_{\mathtt{Add}}\rfloor^{\mathtt{Inc}}\ \mathtt{y}$	$\square$
$\overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc}$	${}_{\mathtt{Inc}}\lceil^{\mathtt{A}}(\overset{\curvearrowleft}{\mathtt{y}}, \sharp \sqsupseteq \mathtt{y} :: \mathtt{A})^{\mathtt{A}}\rceil_{\mathtt{Inc}}\ \mathtt{y}$	$\square$
$\overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc}$	$\overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} = \overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc}$	$\square$
$\mathtt{y} :: \mathtt{A}$	${}^{\mathtt{A}}\lfloor_{\mathtt{Add}}(\mathtt{b} \sqsubseteq \overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} :: \mathtt{Add})\ {}_{\mathtt{Add}}\rfloor_{\mathtt{A}}\ \mathtt{y}$	$\square$
$\mathtt{y} :: \mathtt{A}$	${}^{\mathtt{A}}\lfloor_{\mathtt{Inc}}(\mathtt{z} \sqsubseteq \overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc})\ {}_{\mathtt{Inc}}\rfloor^{\mathtt{A}}\ \mathtt{y}$	$\square$
$\mathtt{y} :: \mathtt{A}$	${}_{\mathtt{A}}\lceil^{\mathtt{Main}}(\mathtt{i} \sqsupseteq \mathtt{i} :: \mathtt{Main})^{\mathtt{Main}}\rceil_{\mathtt{A}}\ \mathtt{y}$	$\square$
$\mathtt{y} :: \mathtt{A}$	$\mathtt{y} = \mathtt{y} :: \mathtt{A}$	$\square$
$\overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} :: \mathtt{Add}$	${}_{\mathtt{Add}}\lceil^{\mathtt{Inc}}(\overset{\curvearrowleft}{\mathtt{z}}, \mathtt{i} \sqsupseteq \overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc})\ {}^{\mathtt{Inc}}\rceil_{\mathtt{Add}}\ \mathtt{y}$	$\square$
$\overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} :: \mathtt{Add}$	${}_{\mathtt{Add}}\lceil^{\mathtt{A}}(\overset{\curvearrowleft}{\mathtt{x}}, \overset{\curvearrowleft}{\mathtt{y}} \sqsupseteq \mathtt{y} :: \mathtt{A})^{\mathtt{A}}\rceil_{\mathtt{Add}}\ \mathtt{y}$	$\square$
$\overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} :: \mathtt{Add}$	$\overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} = \overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} :: \mathtt{Add}$	$\square$
$\overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc}$	${}^{\mathtt{Inc}}\lfloor_{\mathtt{Add}}(\overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} \sqsubseteq \overset{\curvearrowleft}{\mathtt{a}}, \overset{\curvearrowleft}{\mathtt{b}} :: \mathtt{Add})\ {}_{\mathtt{Add}}\rfloor^{\mathtt{Inc}}\ \mathtt{y}$	$\square$
$\overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc}$	${}_{\mathtt{Inc}}\lceil^{\mathtt{A}}(\overset{\curvearrowleft}{\mathtt{y}}, \sharp \sqsupseteq \mathtt{y} :: \mathtt{A})^{\mathtt{A}}\rceil_{\mathtt{Inc}}\ \mathtt{y}$	$\square$
$\overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc}$	$\overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} = \overset{\curvearrowleft}{\mathtt{z}}, \overset{\curvearrowleft}{\mathtt{i}} :: \mathtt{Inc}$	$\square$
$\mathtt{i} :: \mathtt{Main}$	${}^{\mathtt{Main}}\lfloor_{\mathtt{A}}(\mathtt{y} \sqsubseteq \mathtt{y} :: \mathtt{A})\ {}_{\mathtt{A}}\rfloor^{\mathtt{Main}}\ \mathtt{i}$	$\square$
$\mathtt{i} :: \mathtt{Main}$	${}^{\mathtt{Main}}\lfloor_{\mathtt{B}}(\sharp \sqsubseteq \emptyset :: \mathtt{B})\ {}_{\mathtt{B}}\rfloor^{\mathtt{Main}}\ \mathtt{i}$	$\square$
$\mathtt{i} :: \mathtt{Main}$	$\mathtt{i} = \mathtt{i} :: \mathtt{Main}$	$\square$

Fig. 5. Program slicing as term slicing - the fixpoint iterations.

naïve and the *savvy* methods. In order to achieve a better degree of minimality we have to apply abstractions on the data-flow-augmented slicing criterion.

3.2 System description

We briefly present in this section our prototype which is implemented in Maude[2]. The source code is available at `http://maude.sip.ucm.es/slicing/`. A key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [2, Chapter 14]. We have used these features to implement a tool that receives a set of definitions, a sort where the computations take place, and a set of slicing variables. However, the current prototype has a number of limitations. First, it follows the naïve approach which generates bigger slices in general. However, the savvy method requires only a change in the structure of the slicing criterion (from set of variables to dependency relation between variables). Note also that, although our aim is to build a generic tool, the current version relies on some assumptions. Namely, the function definitions are put together by means of an associative operator, and each definition contains the function name, the parameters, and the body, in this order. Finally, the user has to give also the rules responsible for context-update while the parameter passing operators $\lfloor \_ \_ \rfloor$ and $\lceil \_ \_ \rceil$ are particularized here to an all-parameters-ordered-pass-by-reference pattern.

To use the tool we have to introduce `ESt`, the sort for the mapping between variables and values, and `RWBUF`, the sort for the read/write buffer. Similarly, we indicate that `CallF` is the rule for context-update:

```
Maude> (set side-effect sorts ESt RWBUF .)
ESt RWBUF selected as side effect sorts.
Maude> (set context-update rules CallF .)
CallF selected as context-update rules.
```

We can now start the slicing process by indicating that `Statement` is the sort for instructions, `myFuns` is a constant standing for the definition of the functions `Main`, `A`, `Add`, and `Inc`, and `z` is the slicing variable. The tool displays the relevant variables and the sliced code for each function as:

```
Maude> (islice Statement with defs myFuns wrt z .)
The variables to slice 'Inc are {z}
'Inc(z){_ ;_ ; Call 'Add(z,_)}
...
```

4 Concluding Remarks and Ongoing Work

The formal language definitions based on the rewriting logic framework support program executability and create the premises for further development of program analyzers. In this paper we have presented a generic algorithm for interprocedural slicing based on results of meta-level analysis of the language semantics. In summary, the slicing prerequisites are: side-effect and context-update

language constructs with data flow information for the side-effect constructs and parameter passing patterns for the context-update constructs. The actual program slicing computation, presented in the current work, is done through term slicing and is meant to set the aforementioned set of prerequisites. This work complements the recent advances in semantics-constructed tools for debugging [15], automated testing [12], and program analysis [14].

From the prototype point of view, we plan to improve it by adding the savvy slicing method which increases the precision of the slicing result. We also plan to investigate the automatic inference of the newly identified slicing prerequisites, i.e., meta-analysis for context-updates deduction and parameter passing pattern inference. Alos, we have to further develop the already existing side-effect extraction with data flow information. Finally, we aim to develop the method for language semantics defined in Maude but also in $\mathbb{K}$ [16].

References

1. M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Backward trace slicing for conditional rewrite theories. In *LPAR*, pages 62–76, 2012.
2. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
3. C. Ellison and G. Rosu. An executable formal semantics of c with applications. In *POPL*, pages 533–544, 2012.
4. A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal analysis of java programs in javafan. In *CAV*, pages 501–505, 2004.
5. S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP*, pages 253–267, 2007.
6. M. Harman and S. Danicic. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability*, 5:143–162, 1995.
7. M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Wiley & Sons, 1990.
8. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, 1988.
9. R. Jhala and R. Majumdar. Path slicing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, PLDI 2005*, pages 38–47. ACM Press, 2005.
10. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theor. Comput. Sci.*, 285(2):121–154, 2002.
11. J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.
12. A. Riesco. Using semantics specified in Maude to generate test cases. In A. Roychoudhury and M. D'Souza, editors, *Proceedings of the 9th International Colloquium on Theoretical Aspects of Computing, ICTAC 2012*, volume 7521 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2012.
13. A. Riesco. Using big-step and small-step semantics to perform declarative debugging. In M. Codish and E. Sumii, editors, *Proceedings of the 12th International Symposium on Functional and Logic Programming, FLOPS 2014*, Lecture Notes in Computer Science. Springer, 2014.

14. A. Riesco, I. M. Asavoae, and M. Asavoae. A generic program slicing technique based on language definitions. In N. Martí-Oliet and M. Palomino, editors, *Proceedings of the 21st International Workshop on Algebraic Development Techniques, WADT 2012*, volume 7841 of *Lecture Notes in Computer Science*, pages 248–264. IFIP International Federation for Information Processing, 2013.
15. A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. Declarative debugging of rewriting logic specifications. *Journal of Logic and Algebraic Programming*, 2011. To appear.
16. T. F. Şerbănuţă, G. Ştefănescu, and G. Roşu. Defining and executing P systems with structured data in K. In D. W. Corne, P. Frisco, G. Paun, G. Rozenberg, and A. Salomaa, editors, *Proceedings of the 9th International Workshop on Membrane Computing, WMC 2008, Revised Selected and Invited Papers*, volume 5391 of *Lecture Notes in Computer Science*, pages 374–393. Springer, 2009.
17. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis*, pages 189–233, 1981.
18. J. Silva and O. Chitil. Combining algorithmic debugging and program slicing. In *Proceedings of the 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming, PPDP 2006*, pages 157–166. ACM Press, 2006.
19. C. Tian, M. Feng, and R. Gupta. Speculative parallelization using state separation and multiple value prediction. In *Proceedings of the 2010 international symposium on Memory management, ISMM 2010*, pages 63–72. ACM Press, 2010.
20. A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67:226–293, 2006.
21. M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449. IEEE Press, 1981.

Infinite-State Model Checking of LTLR Formulas Using Narrowing

Kyungmin Bae and José Meseguer

Department of Computer Science,
University of Illinois at Urbana-Champaign, Urbana IL 61801
{kbae4,meseguer}@cs.uiuc.edu

Abstract. The linear temporal logic of rewriting (LTLR) is a simple extension of LTL that adds *spatial action patterns* to the logic, expressing that an action described by a rewrite rule has been performed. Although the theory and algorithms of LTLR for finite-state model checking are well-developed [2], no theoretical foundations have yet been developed for infinite-state LTLR model checking. The main goal of this paper is to develop such foundations for *narrowing-based* LTLR model checking. A key theme in this paper is the systematic relationship, in the form of a simulation with remarkably good properties, between the concrete state space and the symbolic state space. A related theme is the use of additional state space reduction methods, such as folding and equational abstractions, that can in some cases yield a finite symbolic state space.

1 Introduction

This paper further develops previous efforts to use rewriting logic and narrowing to perform symbolic model checking of infinite-state systems. Those efforts have gradually increased the expressiveness of the properties that can be verified, first focusing on reachability analysis [16] and then expanding the range to general LTL formulas [1,6]. It is by now clear that state-based temporal logics are not expressive enough to deal with properties involving events, such as message sends and receives; and that the temporal logic of rewriting [14] is a perfect match—at the level of property specification—for rewriting logic—at the level of system specification—so that both can be used seamlessly as a tandem for model checking. For finite-state systems, the authors have developed model checkers that demonstrate the power and usefulness of this tandem of logics [2]. The question asked and positively answered in this paper is: can properties of a rewrite theory $\mathcal{R}$ expressed in the *linear temporal logic of rewriting* (LTLR) [14] be model checked symbolically by narrowing under reasonable assumptions?

The answer to this question is nontrivial, because of a difficulty which can be best explained by briefly recalling how narrowing-based reachability analysis and LTL model checking are performed for a rewrite theory $\mathcal{R}$. For reachability analysis, *any* non-variable term t, symbolically denoting a typically infinite set of concrete state instances, can be narrowed to try to reach an instance of a goal pattern term g. However, for LTL model checking, *not all such terms* t denote

states in the symbolic state space. The reason is that LTL formulas have a set AP of state propositions, but for a symbolic term t such propositions may not be defined: different term instances of t may satisfy different state propositions. The solution proposed in [1,6] is to *specialize* t to most general instances $t_1, \ldots, t_n$ for which all state propositions in AP are either true or false. If the equations defining such propositions have the finite variant property, this can be done by variant narrowing [1,6]. Therefore, narrowing-based LTL model checking symbolically explores the state space of all such *AP-instantiated symbolic terms*.

Suppose that we now want to perform not just LTL model checking but symbolic LTLR model checking, and that our formula φ involves both state propositions in AP and spatial action patterns. For example, a spatial action pattern $l(\theta)$ can appear in φ, stating that a rule $l : q \longrightarrow r$ has been performed with an instantiation that further specializes the substitution θ. As part of the model checking verification of φ we may reach a symbolic state t where we need to check whether the action specified by $l(\theta)$ can be performed. This check will succeed if t can be narrowed with a rule l and a substitution σ such that θ is an *instance* of σ. However, σ can be *incomparable* to θ in general; that is, σ may have instances for which this property holds, and other instances for which it *definitely fails*. This is analogous to the lack of AP-instantiation discussed above for narrowing-based LTL model checking. Let ACT be the set of spatial action patterns we are using, so that, say, $l(\theta) \in ACT$. Our problem is that the symbolic transitions in the LTLR state space need to be *ACT-instantiated*.

Lack of ACT-instantiations is a subtler problem than lack of AP-instantiation. After all, state propositions in AP are equationally defined as Boolean predicates *in both their positive and negative cases*, so that variant narrowing can automate AP-instantiation. The problem of ACT-instantiation has to do with effectively characterizing the *negative cases* in which an action pattern does *not* hold. This turns out to be closely related to the problem of computing *complement patterns* of a pattern term; e.g., for a pattern $l(\theta)$, terms $u_1, \ldots, u_k$ such that any ground term is an instance of *exactly one term* in the set $\{l(\theta), u_1, \ldots, u_k\}$. Not all terms have such complements. For example, for an unsorted singnature with constant 0, unary s, and free binary f, the term $f(x, x)$ has *no* such complements. However, effective methods have been developed to check when a term t has complements and to compute them, e.g., [8,9,12]. Under appropriate assumptions, they can provide a method to solve the ACT-instantiation problem.

Having identified conditions under which the state space for narrowing-based LTRL model checking can be built, the rest of the paper develops the theoretical foundations of narrowing-based LTLR model checking. A key theme in such foundations is the systematic relationship between concrete and symbolic states. This takes the form of a simulation relation from concrete to symbolic states that preserves both state propositions and spatial action patterns. A related theme is the use of additional state space reduction methods, such as folding and equational abstractions, that can in some cases yield a finite symbolic state space. How these foundations can be used in practice to prove nontrivial LTLR properties of infinite-state systems is illustrated with a running example.

2 Preliminaries

Rewriting Logic. An order-sorted signature is a triple $\Sigma = (S, \leq, \Sigma)$ with poset of sorts $(S, \leq)$ and operators $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in S^* \times S}$ typed in $(S, \leq)$. The set $\mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$ denotes the set of Σ-terms of sort s over $\mathcal{X}$ an infinite set of S-sorted variables, and $\mathcal{T}_{\Sigma,\mathsf{s}}$ denotes the set of ground Σ-terms of sort s. We assume that $\mathcal{T}_{\Sigma,\mathsf{s}} \neq \emptyset$ for each sort s in Σ. *Positions* in a term t represent tree positions when t is parsed as a tree, and the replacement in t of a subterm at a position p by another term u is denoted by $t[u]_p$. A *substitution* $\sigma : \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X})$ is a function that maps variables to terms of the same sort, and is homomorphically extended to $\mathcal{T}_\Sigma(\mathcal{X})$ in a natural way. The *domain* of σ is a finite subset $dom(\sigma) \subseteq \mathcal{X}$, where $\sigma x = x$ for any $x \notin dom(\sigma)$. The restriction of σ to $Y \subseteq \mathcal{X}$ is the substitution $\sigma|_Y$ such that $\sigma|_Y(x) = \sigma(x)$ if $x \in Y$, and $\sigma|_Y(x) = x$ otherwise.

A rewrite theory is a formal specification of a concurrent system [13]. To apply narrowing-based methods, we consider *unconditional order-sorted rewrite theories* $\mathcal{R} = (\Sigma, E, R)$, where: (i) (Σ, E) is an equational theory with Σ an order-sorted signature and E a set of equations, specifying the system's states as the initial algebra $\mathcal{T}_{\Sigma/E}$ (i.e., each state is an E-equivalence class $[t]_E \in \mathcal{T}_{\Sigma/E}$ of ground terms); and (ii) R is a set of *rewrite rules* of the form $l : q \longrightarrow r$ with label l and Σ-terms $q, r \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$, specifying the system's transitions as a *one-step rewrite* $t[l(\theta)]_p : [t[\theta q]_p]_E \longrightarrow_{\mathcal{R}} [t[\theta r]_p]_E$ from a state $[t[\theta q]_p]_E \in \mathcal{T}_{\Sigma/E}$ containing a substitution instance θq to the state $[t[\theta r]_p]_E \in \mathcal{T}_{\Sigma/E}$ in which θq has been replaced by θr, where $t[l(\theta)]_p$ is called a *one-step proof term*.

We also require $\mathcal{R} = (\Sigma, E, R)$ being *topmost* for narrowing-based methods. That is, there is sort State at the top of one of the connected component of $(S, \leq)$ such that: (i) for each rule $l : q \longrightarrow r \in R$, both q and r have the top sort State; and (ii) no operator in Σ has State or any of its subsorts as an argument sort. This ensures that all rewrites with rules in R must take place at the top of the term. In practice, many concurrent systems, including object-oriented systems and communication protocols, can be specified by topmost rewrite theories [16].

We can associate to $\mathcal{R}$ a corresponding Kripke structure for LTL model checking. A *Kripke structure* is a 4-tuple $\mathcal{K} = (S, AP, \mathcal{L}, \longrightarrow_{\mathcal{K}})$ with S a set of *states*, AP a set of atomic *state propositions*, $\mathcal{L} : S \to \mathcal{P}(AP)$ a *state-labeling function*, and $\longrightarrow_{\mathcal{K}} \subseteq S \times S$ a *total transition relation* in which every state $s \in S$ has a next state $s' \in S$ with $s \longrightarrow_{\mathcal{K}} s'$. A state proposition is defined as a term of sort Prop, whose meaning is defined by equations using the auxiliary operator $\_\models\_ : \mathsf{State}\ \mathsf{Prop} \to \mathsf{Bool}$. By definition, $p \in \mathcal{T}_{\Sigma/E,\mathsf{Prop}}$ is satisfied on a state $[t]_E$ iff $(t \models p) =_E \mathit{true}$. We assume that sort Bool has two constants *true* and *false* with $\mathit{true} \neq_E \mathit{false}$ and any $t \in \mathcal{T}_{\Sigma,\mathsf{Bool}}$ is provably equal to either *true* or *false*.

Definition 1. *Given $\mathcal{R} = (\Sigma, E, R)$ and a set $AP \subseteq \mathcal{T}_{\Sigma/E,\mathsf{Prop}}$ defined by E, the corresponding Kripke structure is $\mathcal{K}(\mathcal{R})_{AP} = (\mathcal{T}_{\Sigma/E,\mathsf{State}}, AP, \mathcal{L}_E, \longrightarrow_{\mathcal{R}})$,[1] where $\mathcal{L}_E([t]_E) = \{p \in AP \mid (t \models p) =_E \mathit{true}\}$.*

[1] Since $\longrightarrow_{\mathcal{R}}$ needs to be total, we also assume that $\mathcal{R}$ is deadlock-free. Note that $\mathcal{R}$ can be easily transformed into an equivalent deadlock-free theory [15].

Linear Temporal Logic of Rewriting. The *linear temporal logic of rewriting* (LTLR) is a state/event extension of LTL with *spatial action patterns* [2]. An LTLR formula φ may include spatial action patterns $\delta_1, \ldots, \delta_n$ as well as state propositions $p_1, \ldots, p_m$, and therefore may describe properties involving both states and events. Given a set of state propositions AP and a set of spatial action patterns ACT, the syntax of LTLR is defined by $\varphi ::= p \mid \delta \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi \,\mathbf{U}\, \varphi$, where $p \in AP$ and $\delta \in ACT$. Other operators can be defined by equivalences, e.g., $\Diamond\varphi \equiv \mathit{true}\,\mathbf{U}\,\varphi$ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$.

Spatial action patterns describe properties of one-step rewrites by defining a set of matching one-step proof terms. For example, a pattern l describes that a rule with label l is applied, and a pattern $l(\theta)$ describes that a rule with label l is applied and the related variable instantiation is a further instantiation of the substitution θ [2,14]. In a similar way that state propositions of LTL are defined by equations, the matching relation $\models$ between a one-step proof term γ and a spatial action pattern δ can be defined by equations using the auxiliary operator $\_\models\_ : \mathsf{ProofTerm}\ \mathsf{Action} \to \mathsf{Bool}$, where $\gamma \models \delta \iff (\gamma \models \delta) =_E \mathit{true}$.

The semantics of an LTLR formula is defined on a *labeled Kripke structure* (LKS), an extension of a Kripke structure with transition labels [2,3]. An LKS is a 5-tuple $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \longrightarrow_{\bar{\mathcal{K}}})$ with S a set of *states*, AP a set of *state propositions*, $\mathcal{L} : S \to \mathcal{P}(AP)$ a *state-labeling function*, ACT a set of *spatial action patterns*, and $\longrightarrow_{\bar{\mathcal{K}}} \subseteq S \times \mathcal{P}(ACT) \times S$ a total *labeled transition relation*. A *path* (π, α) is a pair of functions $\pi : \mathbb{N} \to S$ and $\alpha : \mathbb{N} \to \mathcal{P}(ACT)$ such that $\pi(i) \xrightarrow{\alpha(i)}_{\bar{\mathcal{K}}} \pi(i+1)$, and $(\pi, \alpha)^k$ denotes the suffix of (π, α) beginning at position k such that $(\pi, \alpha)^k = (\pi \circ s^k, \alpha \circ s^k)$ with s the successor function.

We can associate to a rewrite theory $\mathcal{R}$ a corresponding LKS $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$ for LTLR model checking, provided that the state propositions AP and the spatial action patterns ACT are defined by its equations.

Definition 2. *Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, sets $AP \subseteq \mathcal{T}_{\Sigma/E,\mathsf{Prop}}$ and $ACT \subseteq \mathcal{T}_{\Sigma/E,\mathsf{Action}}$ defined by E, the corresponding LKS is*

$$\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT} = (\mathcal{T}_{\Sigma/E,\mathsf{State}}, AP, \mathcal{L}_E, ACT, \longrightarrow_{\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}}),$$

where $\mathcal{L}_E([t]_E) = \{p \in AP \mid (t \models p) =_E \mathit{true}\}$, and $[t]_E \xrightarrow{A}_{\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}} [t']_E$ iff $\gamma : [t]_E \longrightarrow_{\mathcal{R}} [t']_E$ and $A = \{\delta \in ACT \mid (\gamma \models \delta) =_E \mathit{true}\}$.

Given an LTLR formula φ and an initial state $s_0 \in S$, the satisfaction relation $\bar{\mathcal{K}}, s_0 \models \varphi$ holds iff for each path (π, α) of $\bar{\mathcal{K}}$ beginning at s_0, the path satisfaction relation $\bar{\mathcal{K}}, (\pi, \alpha) \models \varphi$ holds, which is defined inductively as follows:

- $\bar{\mathcal{K}}, (\pi, \alpha) \models p$ iff $p \in \mathcal{L}(\pi(0))$
- $\bar{\mathcal{K}}, (\pi, \alpha) \models \delta$ iff $\delta \in \alpha(0)$
- $\bar{\mathcal{K}}, (\pi, \alpha) \models \neg\varphi$ iff $\bar{\mathcal{K}}, (\pi, \alpha) \not\models \varphi$
- $\bar{\mathcal{K}}, (\pi, \alpha) \models \varphi \wedge \varphi'$ iff $\bar{\mathcal{K}}, (\pi, \alpha) \models \varphi$ and $\bar{\mathcal{K}}, (\pi, \alpha) \models \varphi'$
- $\bar{\mathcal{K}}, (\pi, \alpha) \models \bigcirc\varphi$ iff $\bar{\mathcal{K}}, (\pi, \alpha)^1 \models \varphi$
- $\bar{\mathcal{K}}, (\pi, \alpha) \models \varphi \,\mathbf{U}\, \varphi'$ iff $\exists k \geq 0.\ \bar{\mathcal{K}}, (\pi, \alpha)^k \models \varphi'$, $\forall 0 \leq i < k.\ \bar{\mathcal{K}}, (\pi, \alpha)^i \models \varphi$.

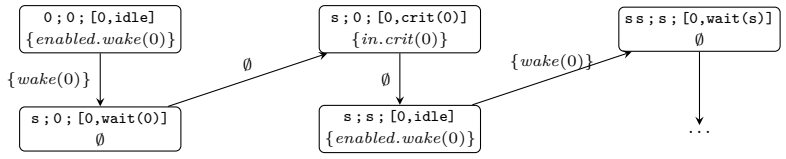


Fig. 1. A path from `0 ; 0 ; [0,idle]` in the LKS $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$ for the bakery protocol.

Example. We present a topmost rewrite theory $\mathcal{R}$ specifying Lamport's bakery protocol for mutual exclusion (adapted from [1,6]), and its corresponding LKS $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$. Each state of the system has the form

$$n\ ;\ m\ ;\ [i_1, d_1] \ldots [i_n, d_n],$$

given by the operator $\_;\_;\_ : \mathsf{Nat\ Nat\ ProcSet} \rightarrow \mathsf{State}$, where n is the current number in the bakery's number dispenser, m is the number currently being served, and the $[i_1, d_1] \ldots [i_n, d_n]$ are a set of customer processes, each with a name i_l and in a *mode* d_l. A mode can be *idle* (not yet picked a number), *wait*(n) (waiting with number n), or *crit*(n) (being served with number n). The behavior is specified by the following *topmost* rewrite rules in the Maude language:

```
rl [wake]: N ; M ; [I,idle] PS  =>  s N ; M ; [I,wait(N)] PS .
rl [crit]: N ; M ; [I,wait(M)] PS => N ; M ; [I,crit(M)] PS .
rl [exit]: N ; M ; [I,crit(M)] PS => N ; s M ; [I,idle] PS .
```

where natural numbers are modeled as multisets of s with the multiset union operator `__` (empty syntax) and the empty multiset `0` (e.g., $0 = \mathtt{0}$, and $3 = \mathtt{s\ s\ s}$).

We are interested in verifying the liveness property "*process 0 is eventually served*," under the fairness assumption "*if process 0 can eventually pick a number forever, it must pick a number infinitely often*," expressed as the LTLR formula

$$(\Diamond\Box\mathit{enabled.wake}(0) \rightarrow \Box\Diamond\mathit{wake}(0)) \rightarrow \Diamond\mathit{in.crit}(0),$$

where the spatial action pattern *wake*(0) holds if the *wake* rule is applied for process 0 (i.e., the variable `I` in the *wake* rule is matched to the term `0`), the state proposition *enabled.wake*(0) holds in a state where process 0 is idle, and the state proposition *in.crit*(0) holds in a state where process 0 is being served (see [1] for the mutual exclusion property).

For the set of state propositions $AP = \{\mathit{in.crit}(0), \mathit{enabled.wake}(0)\}$ and the set of spatial action patterns $ACT = \{\mathit{wake}(0)\}$, we can construct the related LKS $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$ for the bakery protocol specification $\mathcal{R}$. For example, given the initial state `0 ; 0 ; [0,idle]`, we obtain the infinite path in Figure 1 within $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$ that contains an infinite number of different states. Notice that this system is infinite-state since: (i) the counters n and m are unbounded; and (ii) the number of customer processes is unbounded.

3 Narrowing-based LTLR Model Checking

Narrowing [10,11] generalizes term rewriting by allowing free variables in terms and by performing unification instead of matching. An *E-unifier* of $t = t'$ is a substitution σ such that $\sigma t =_E \sigma t'$ and $dom(\sigma) \subseteq vars(t) \cup vars(t')$, and $CSU_E(t = t')$ denotes a *complete set of E-unifiers* in which any E-unifier ρ of $t = t'$ has a more general substitution $\sigma \in CSU_E(t = t')$, i.e., $(\exists \eta)\ \rho =_E \eta \circ \sigma$. We assume that there exists a finitary E-unification procedure to find a *finite* complete set $CSU_E(t = t')$ of E-unifiers (e.g., there exists a finitary E-unification procedure if E has the *finite variant property* as explained in [5,7]).

Definition 3. *Given a* topmost *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$*, each rewrite rule* $l : q \longrightarrow r \in R$ *specifies a topmost* narrowing step $t \rightsquigarrow_{l,\sigma,\mathcal{R}} t'$ *(or* $t \rightsquigarrow_{\mathcal{R}} t'$*) iff there exists an E-unifier* $\sigma \in CSU_E(t = q)$ *such that* $t' = \sigma r$*.*

For LTL model checking we can associate to $\mathcal{R} = (\Sigma, E, R)$ a corresponding *logical Kripke structure* $\mathcal{N}(\mathcal{R})_{AP}$ [6]. The states of $\mathcal{N}(\mathcal{R})_{AP}$ are AP-instantiated elements of $\mathcal{T}_{\Sigma/E}(\mathcal{X})_{\mathsf{State}}$ and its transitions are specified by $\rightsquigarrow_{\mathcal{R}}$. A state of $\mathcal{N}(\mathcal{R})_{AP}$ is not a *concrete state*, but a *state pattern* $t(x_1, \ldots, x_n)$ with *logical variables* $x_1, \ldots, x_n$, representing the set of all concrete states $[\theta t]_E$ that are its *ground instances*. Such a logical Kripke structure $\mathcal{N}(\mathcal{R})_{AP}$ can be considered as an abstraction of the concrete system $\mathcal{K}(\mathcal{R})_{AP}$; i.e., for an LTL formula φ and a state pattern t, $\mathcal{N}(\mathcal{R})_{AP}, [t]_E \models \varphi$ implies $(\forall \theta : \mathcal{X} \to \mathcal{T}_\Sigma)\ \mathcal{K}(\mathcal{R})_{AP}, [\theta t]_E \models \varphi$. Generalizing such narrowing-based LTL model checking, this section presents narrowing-based LTLR model checking for infinite-state systems.

One-Step Proof Terms for Narrowing. Spatial action patterns for rewriting define their matching one-step proof terms, representing the corresponding one-step rewrites. For a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$, one-step proof terms have the form $l(\theta)$, indicating that a rule $l : q \longrightarrow r \in R$ has been applied with a substitution θ (at the top position of the term), where $dom(\theta) \subseteq vars(q) \cup vars(r)$.

In order to define spatial action patterns for narrowing steps, we also need to have an appropriate notion of one-step proof terms for narrowing. Consider a topmost narrowing step $t \rightsquigarrow_{l,\sigma,\mathcal{R}} t'$ using a rule $l : q \longrightarrow r$. Intuitively, the rule label l and the restriction of the substitution σ to the variables in the rule[2] give the one-step proof term for the narrowing step $t \rightsquigarrow_{l,\sigma,\mathcal{R}} t'$.

Definition 4. *Given a* topmost *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$*, for a topmost narrowing step* $t \rightsquigarrow_{l,\sigma,\mathcal{R}} t'$ *using a rule* $l : q \longrightarrow r$*, its one-step proof term is given by* $l(\sigma|_{vars(q) \cup vars(r)})$*, often denoted by* $l(\sigma_l)$*.*

The following lemma implies that a one-step proof term $l(\sigma_l)$ for narrowing faithfully captures its corresponding one-step proof terms $l(\theta)$ for rewriting, in the sense that $\theta =_E \eta \circ \sigma_l$ for some substitution η. This lemma is adapted from the soundness and completeness results of topmost narrowing in [16].

[2] Since one-step proof terms for rewriting only contain variables in rules, we restrict one-step proof terms for narrowing in the same way.

Lemma 1. *Given a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$, for a non-variable term u and a substitution ρ, assuming no variable in u appears in the rules R, $(\exists t', \theta)\ l(\theta) : \rho u \longrightarrow_{\mathcal{R}} t'$ iff $(\exists u', \sigma, \eta)\ u \rightsquigarrow_{l,\sigma,\mathcal{R}} u' \wedge \rho|_{vars(u)} =_E (\eta \circ \sigma)|_{vars(u)}$, where $\theta =_E (\eta \circ \sigma)|_{dom(\theta)}$ and $t' =_E \eta u'$.*

Proof. ($\Rightarrow$) Suppose that $l(\theta) : \rho u \longrightarrow_{\mathcal{R}} t'$ for a topmost rule $l : q \longrightarrow r$, where $dom(\theta) \subseteq vars(q) \cup vars(r)$. Then, $\theta q =_E \rho u$ and $t' = \theta r$. Since *no* variable in u appears in $l : q \longrightarrow r$, we have $dom(\theta) \cap vars(u) = \emptyset$. Thus, we can define the substitution $\theta \cup \rho|_{vars(u)}$ with domain $dom(\theta) \cup vars(u)$ such that $(\theta \cup \rho|_{vars(u)})|_{dom(\theta)} = \theta$ and $(\theta \cup \rho|_{vars(u)})|_{vars(u)} = \rho|_{vars(u)}$. Since $\theta \cup \rho|_{vars(u)}$ is an E-unifier of $q = u$, there exist substitutions $\sigma \in CSU_E(u = q)$ and η' satisfying $(\theta \cup \rho|_{vars(u)})|_{vars(q)\cup vars(u)} =_E \eta' \circ \sigma$ with domain $vars(q) \cup vars(u)$. Therefore, $u \rightsquigarrow_{l,\sigma,\mathcal{R}} u'$ for $u' = \sigma r$. Next, let η be the extended substitution such that $\eta x = \eta' x$ if $x \in vars(q) \cup vars(u)$, and $\eta x = \theta x$ otherwise. Then, $\rho|_{vars(u)} =_E (\eta \circ \sigma)|_{vars(u)}$ and $\theta =_E (\eta \circ \sigma)|_{dom(\theta)}$, since $dom(\theta) \cap vars(u) = \emptyset$ and $dom(\theta) \subseteq vars(q) \cup vars(r)$. Furthermore, $t' = \theta r =_E (\eta \circ \sigma) r = \eta u'$. ($\Leftarrow$) Suppose that $u \rightsquigarrow_{l,\sigma,\mathcal{R}} u'$ and $\rho|_{vars(u)} =_E (\eta \circ \sigma)|_{vars(u)}$. Then, for a topmost rule $l : q \longrightarrow r$, $\sigma \in CSU_E(u = q)$ and $u' = \sigma r$. Since $\sigma u =_E \sigma q$ and $(vars(q) \cup vars(r)) \cap vars(u) = \emptyset$, we have $l(\sigma|_{vars(q)\cup vars(r)}) : \sigma u \longrightarrow_{\mathcal{R}} u'$. Thus, we have $l(\eta \circ \sigma|_{vars(q)\cup vars(r)}) : (\eta \circ \sigma) u \longrightarrow_{\mathcal{R}} \eta u'$, where $(\eta \circ \sigma) u =_E \rho u$, since rewrites are stable under substitutions. □

Equational Definition of State/Event Predicates. The semantics of a spatial action pattern can be defined by means of equations using the auxiliary operator `_|=_ : ProofTerm Action -> Bool` [2]. By definition, $\delta \in \mathcal{T}_{\Sigma/E,\mathsf{Action}}$ is matched to a one-step proof term γ iff $(\gamma \models \delta) =_E \mathit{true}$. For a topmost rewrite theory $\mathcal{R}$, a one-step proof term $l(\theta)$ can be represented as a term

$$\{'l\ :\ 'x_1 \backslash \theta x_1\ ;\ \ldots\ ;\ 'x_m \backslash \theta x_m\}$$

of sort `ProofTerm` using the operator `{_:_} : Qid Substitution -> ProofTerm`, where $'l, 'x_1, \ldots, 'x_m$ are quoted identifiers of sort `Qid` and $'x_1 \backslash \theta x_1; \ldots; 'x_m \backslash \theta x_m$ is a semicolon separated set of variable assignments. For the bakery example, a topmost narrowing step from the term `N ; N ; [0,idle]` by the *wake* rule gives the one-step proof term `{'wake : 'N \ N ; 'M \ N ; 'I \ 0 ; 'PS \ none}`.

For narrowing-based model checking we further require that there exists a finitary E-unification procedure. If a spatial action pattern δ is identified by a one-step proof term *pattern* u_δ (i.e., $(\gamma \models \delta) =_E \mathit{true}$ iff γ is an instance of u_δ),[3] and if u_δ *has complement patterns* $u_1, \ldots, u_k$ (i.e., any ground one-step proof term is an instance of exactly one term in $\{u_\delta, u_1, \ldots, u_k\}$), then δ can be defined by the equations: $u_\delta \models \delta = \mathit{true}$, $u_1 \models \delta = \mathit{false}$, $\ldots$, $u_k \models \delta = \mathit{false}$. Since the right-hand sides are all constants, these equations have the finite variant property, and therefore they provide a finitary E-unification algorithm [5,7]. This method can also be applied for "pattern-like" state propositions (see below).

[3] Many spatial action patterns, including l and $l(\theta)$, are identified in this way [2,14].

As mentioned in the introduction, effective methods have been developed to check when a term t has complements and to compute such complement patterns, not only in the free case [12], but also modulo AC and modulo permutative theories [8,9]. Therefore, for unconditional rewrite theories with axioms B such as those used in [8,9,12], we can determine if a one-step proof term pattern u_δ of δ has complements, compute such complement patterns, and define pattern satisfaction of δ by equations. For example, consider the spatial action pattern $wake(0)$ in the bakery example. The positive case can be defined by the following equation, where `SUBST` is a variable of sort Substitution:

```
eq {'wake : 'I \ 0; SUBST} |= wake(0) = true .
```

For the negative cases, $wake(0)$ does *not* hold when the rule label is *not* `'wake` or the value of `'I` is *not* `0`. Therefore, they can be defined by the complement patterns of 0 and `'wake` as follows.

```
eq {'wake : 'I \ s J ; SUBST} |= wake(0) = false .
eq {'crit : SUBST} |= wake(0) = false .
eq {'exit : SUBST} |= wake(0) = false .
```

The use of order-sorted signatures can greatly facilitate the existence of complement patterns that may not exist in an unsorted setting. For example, the unsorted term $y + 0 + 0$ for a signature with a constant 0, a unary s, and an AC symbol $+$ is shown not to have complements in [8], but can be easily shown to have complements when the signature is refined to an order-sorted signature. We illustrate this greater ease of computing complements by using the state propositions $in.crit(0)$ and $enabled.wake(0)$, whose positive cases are defined by the following equations, where `PS` is a variable of sort ProcSet:

```
eq N ; M ; [0,crit(K)] PS |= in.crit(0) = true .
eq N ; M ; [0,idle] PS |= enabled.wake(0) = true .
```

In order to define the negative cases we need to find the complement patterns for `[0,crit(K)] PS` and `[0,idle] PS`. Using subsort relations, we can define sort ModeIdleWait for *idle* and *wait(n)*, ModeWaitCrit for *wait* and *crit(n)*, and ProcSet{N0Nat} for a set of processes with non-zero identifiers as follows:[4]

```
subsorts ModeIdle ModeWait < ModeIdleWait < Mode .
subsorts ModeWait ModeCrit < ModeWaitCrit < Mode .
subsorts N0Nat < Nat .
subsorts Proc{N0Nat} < ProcSet{N0Nat} Proc < ProcSet .
```

The negative cases for the above state propositions can then be defined by the following equations, where the variable `DIW` has sort ModeIdleWait, `DWC` has sort ModeWaitCrit, and `NZPS` has sort ProcSet{N0Nat}:

```
eq N ; M ; [0,DIW] NZPS |= in.crit(0) = false .
eq N ; M ; [0,DWC] NZPS |= enabled.wake(0) = false .
```

[4] Generally, to define the negative cases for $k \in \mathbb{N}$, we can define $k + 2$ subsorts Nat0, ..., Nat$k$, NkNat of sort Nat, where NkNat denotes a number greater than k.

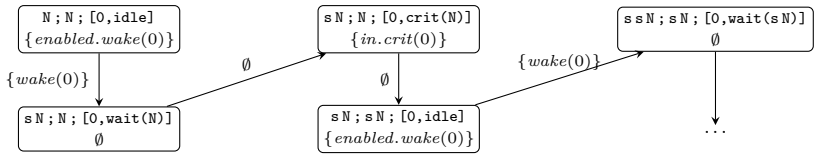


Fig. 2. A path from `N ; N ; [0,idle]` in the LKS $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$ for the bakery protocol.

Narrowing-based LKS. For a set $AP = \{p_1, \ldots, p_n\}$ of state propositions and a set $ACT = \{\delta_1, \ldots, \delta_m\}$ of spatial action patterns defined by E, we can also associate to a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$ a corresponding *narrowing-based logical LKS* $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$. Each state of $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ is a term in which the truth of every state proposition is decided into either *true* or *false*. A transition of $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ is specified by using a topmost narrowing step $\rightsquigarrow_{\mathcal{R}}$, but further instantiated into possibly several transitions so that the truth b_i of each state proposition p_i, where $1 \leq i \leq n$, and the truth b_{n+j} of each spatial action pattern δ_j, where $1 \leq j \leq m$, are decided into *true* or *false*.

Definition 5. *Given a topmost rewrite theory* $\mathcal{R} = (\Sigma, E, R)$*, and* finite *sets* $AP = \{p_1, \ldots, p_n\} \subseteq \mathcal{T}_{\Sigma/E,\mathsf{Prop}}$ *and* $ACT = \{\delta_1, \ldots, \delta_m\} \subseteq \mathcal{T}_{\Sigma/E,\mathsf{Action}}$ *defined by its equations* E*, the narrowing-based* logical LKS *is*

$$\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT} = (N(\mathcal{R})_{AP}, AP, \mathcal{L}_E, ACT, \longrightarrow_{\bar{\mathcal{N}}(\mathcal{R})}),$$

where $\mathcal{L}_E([t]_E) = \{p \in AP \mid (t \models p) =_E true\}$*, and:*

- $[t]_E \in N(\mathcal{R})_{AP}$ *iff* $[t]_E \in \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\mathsf{State}}$*,* $\mathcal{X}$*, and for every* $p \in AP$*, either* $(t \models p) =_E true$ *or* $(t \models p) =_E false$*.*
- $[t]_E \xrightarrow{A}_{\bar{\mathcal{N}}(\mathcal{R})} [t']_E$ *iff there exist a term* u*, a substitution* ζ*, and Boolean values* $b_1, \ldots, b_{n+m} \in \{true, false\}$ *such that*

$$t \rightsquigarrow_{l,\sigma,\mathcal{R}} u \;\wedge\; t' = \zeta u, \;\wedge\; A = \{\delta \in ACT \mid (\zeta(l(\sigma_l)) \models \delta) =_E true\} \;\wedge$$
$$\zeta \in CSU_E\big(\textstyle\bigwedge_{1 \leq i \leq n}(u \models p_i) = b_i \wedge \bigwedge_{1 \leq j \leq m}(l(\sigma_l) \models \delta_j) = b_{n+j}\big)$$

For the bakery example, given the *logical* initial state `N ; N ; [0,idle]`, we obtain within the logical LKS $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ the infinite path in Figure 2, which captures an infinite number of concrete paths in the concrete LKS $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$ starting from each ground instance of `N ; N ; [0,idle]`.

A narrowing-based LKS $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ captures any behavior of the related concrete LKS $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$, in terms of a *simulation relation*. In the following definition we extend the usual notion of a simulation for Kripke structures to one for LKSs, which also takes into account spatial action patterns.

Definition 6. *Given two LKS* $\bar{\mathcal{K}}_i = (S_i, AP, \mathcal{L}_i, ACT, \longrightarrow_{\bar{\mathcal{K}}_i})$*,* $i = 1, 2$*, a binary relation* $H \subseteq S_1 \times S_2$ *is a* simulation *from* $\bar{\mathcal{K}}_1$ *to* $\bar{\mathcal{K}}_2$ *iff: (i) if* $s_1 H s_2$*, then* $\mathcal{L}_1(s_1) = \mathcal{L}_2(s_2)$*, and (ii) if* $s_1 H s_2$ *and* $s_1 \xrightarrow{A}_{\bar{\mathcal{K}}} s_1'$*, there exists* $s_2' \in S_2$ *such that* $s_1' H s_2'$ *and* $s_2 \xrightarrow{A}_{\bar{\mathcal{K}}} s_2'$*. A simulation* H *is a* bisimulation *iff* H^{-1} *is also a simulation, and is* total *iff for any* $s_1 \in S_1$ *there exists* $s_2 \in S_2$ *such that* $s_1 H s_2$*.*

As expected, if an LKS $\bar{\mathcal{K}}_2$ *simulates* $\bar{\mathcal{K}}_1$, then each infinite path in $\bar{\mathcal{K}}_1$ has a corresponding path in $\bar{\mathcal{K}}_2$, as shown in the following lemma.

Lemma 2. *Given a simulation H from an LKS $\bar{\mathcal{K}}_1$ to $\bar{\mathcal{K}}_2$, if $s_1 H s_2$, then for each path (π_1, α) of $\bar{\mathcal{K}}_1$ beginning at s_1, there exists a corresponding path (π_2, α) beginning at s_2 such that $\pi_1(i)\, H\, \pi_2(i)$ for each $i \in \mathbb{N}$.*

Proof. We construct π_2 by induction. Let $\pi_2(0) = s_2$. Clearly, $\pi_1(0)\, H\, \pi_2(0)$. Next, suppose that $\pi_1(k)\, H\, \pi_2(k)$ for some $k \in \mathbb{N}$. Since $\pi_1(k)\, H\, \pi_2(k)$ and $\pi_1(k) \xrightarrow{\alpha(k)}_{\bar{\mathcal{K}}} \pi_1(k+1)$, there exists a state s_2' such that $\pi_1(k+1)\, H\, s_2'$ and $\pi_2(k) \xrightarrow{\alpha(k)}_{\bar{\mathcal{K}}} s_2'$. Then, we choose $\pi_2(k+1) = s_2'$. □

Suppose that $s_0^1 H s_0^2$ for a simulation H from $\bar{\mathcal{K}}_1$ to $\bar{\mathcal{K}}_2$. If there exists a counterexample (π_1, α_1) in $\bar{\mathcal{K}}_1$ starting from s_0^1, then by the above lemma, there exists a corresponding counterexample (π_2, α_2) in $\bar{\mathcal{K}}_2$ starting from s_0^2 such that $\mathcal{L}_1(\pi_1(i)) = \mathcal{L}_2(\pi_2(i))$ and $\alpha_1(i) = \alpha_2(i)$ for each $i \in \mathbb{N}$. Therefore:

Corollary 1. *Given a simulation H from an LKS $\bar{\mathcal{K}}_1$ to $\bar{\mathcal{K}}_2$, if $s_0^1 H s_0^2$, then for any LTLR formula φ, $\bar{\mathcal{K}}_2, s_0^2 \models \varphi$ implies $\bar{\mathcal{K}}_1, s_0^1 \models \varphi$. In particular, if H is a bisimulation, then $\bar{\mathcal{K}}_2, s_0^2 \models \varphi$ iff $\bar{\mathcal{K}}_1, s_0^1 \models \varphi$.*

For a narrowing-based LKS $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$, each logical state is clearly related to a concrete state in $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$ in terms of the *E-subsumption* relation. The E-subsumption $t \preccurlyeq_E t'$ holds iff there exists a substitution σ with $t =_E \sigma t'$, meaning that t' is *more general* than t modulo E.

Lemma 3. *Given a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and sets AP and ACT defined by E, $\preccurlyeq_E$ is a total simulation from $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$ to $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$.*

Proof. Suppose that $[t]_E \xrightarrow{A}_{\bar{\mathcal{K}}(\mathcal{R})} [t']_E$ and $t \preccurlyeq_E u$ for $u \in N(\mathcal{R})_{AP}$. Given $AP = \{p_1, \ldots, p_n\}$ and $ACT = \{\delta_1, \ldots, \delta_m\}$, fix $b_1, b_2, \ldots, b_{n+m} \in \{true, false\}$ such that $b_i =_E (t' \models p_i)$ for $1 \le i \le n$ and $b_{n+j} =_E (l(\theta) \models \delta_j)$ for $1 \le j \le m$. By definition, there is an one-step rewrite $l(\theta) : t \longrightarrow_{\mathcal{R}} t'$. By Lemma 1, there is a narrowing step $u \rightsquigarrow_{l,\sigma,\mathcal{R}} u'$ such that $t' =_E \eta u'$ and $\theta =_E (\eta \circ \sigma)|_{dom(\theta)}$. Thus, there exists $\zeta \in CSU_E(\bigwedge_{1\le i\le n}(u' \models p_i) = b_i \ \wedge\ \bigwedge_{1\le j\le m}(l(\sigma_l) \models \delta_j) = b_{n+j})$. By definition, $[u]_E \xrightarrow{A}_{\bar{\mathcal{N}}(\mathcal{R})} [\zeta u']_E$. Notice that $\bigwedge_{1\le i\le n}\eta\big((u' \models p_i) =_E b_i\big)$ and $\bigwedge_{1\le j\le m}\eta\big((l(\sigma_l) \models \delta_j) =_E b_{n+j}\big)$. Therefore, $\eta \preccurlyeq_E \zeta$, and $t' =_E \eta u \preccurlyeq_E \zeta u'$. □

By Corollary 1, this lemma implies that any LTLR formula φ satisfied in a narrowing-based LKS $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ from a logical state t is also satisfied in the concrete LKS $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$ from each ground instance of t.

In general, $\preccurlyeq_E$ is *not* a bisimulation between $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$ and $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$. For the bakery example, although `0 ; 0 ; [I,wait(0)]` $\preccurlyeq_E$ `N ; M ; PS`$_1$ holds, there exists the transition `N ; M ; PS`$_1$ $\xrightarrow{\{wake(0)\}}_{\bar{\mathcal{N}}(\mathcal{R})}$ `s N ; M ; PS`$_2$ `[0,wait(N)]`, in $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ with the substitution `PS`$_1 \setminus$ `PS`$_2$ `[0,idle]`, but *no corresponding transition* exists from `0 ; 0 ; [I,wait(0)]` in $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$. However, any *finite* path in $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ can be *instantiated* to a corresponding concrete path in $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$ (e.g., the above transition can be instantiated as the transition `0 ; 0 ; [0,idle]` $\xrightarrow{\{wake(0)\}}_{\bar{\mathcal{K}}(\mathcal{R})}$ `s ; 0 ; [0,wait(0)]` in $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$).

Lemma 4. *For a* finite *path* $u_1 \xrightarrow{A_1}_{\bar{\mathcal{N}}(\mathcal{R})} \cdots \xrightarrow{A_{n-1}}_{\bar{\mathcal{N}}(\mathcal{R})} u_n$ *of* $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$, *there is* $t_1 \xrightarrow{A_1}_{\bar{\mathcal{K}}(\mathcal{R})} \cdots \xrightarrow{A_{n-1}}_{\bar{\mathcal{K}}(\mathcal{R})} t_n$ *in* $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$ *with* $t_i \preccurlyeq_E u_i$, $1 \leq i \leq n$.

Proof. Since $u_1 \xrightarrow{A_1}_{\bar{\mathcal{N}}(\mathcal{R})} u_2$, by definition, there are substitutions σ_1 and ζ_1 such that $u_1 \rightsquigarrow_{l_1,\sigma_1,\mathcal{R}} u_2'$ by a topmost rule $l_1 : q_1 \to r_1 \in R$ and $u_2 = \zeta_1 u_2'$. Since $\sigma u_1 =_E \sigma q_1$ and $u_2 = \zeta_1 u_2' = (\zeta_1 \circ \sigma_1) r_1$, $(\zeta_1 \circ \sigma_1) u_1 \longrightarrow_{\mathcal{R}} u_2$. Similarly, $(\zeta_2 \circ \sigma_2) u_2 \longrightarrow_{\mathcal{R}} u_3$, etc. By composing them, $(\zeta_{n-1} \circ \sigma_{n-1} \circ \cdots \circ \zeta_2 \circ \sigma_2 \circ \zeta_1 \circ \sigma_1) u_1 \longrightarrow_{\mathcal{R}} \cdots \longrightarrow_{\mathcal{R}} (\zeta_{n-1} \circ \sigma_{n-1}) u_{n-1} \longrightarrow_{\mathcal{R}} u_n$. Let ρ be a ground substitution instantiating every variable in the path. Then, $(\rho \circ \zeta_{n-1} \circ \sigma_{n-1} \circ \cdots \circ \zeta_2 \circ \sigma_1) u_1 \longrightarrow_{\mathcal{R}} \cdots \longrightarrow_{\mathcal{R}} (\rho \circ \zeta_{n-1} \circ \sigma_{n-1}) u_{n-1} \longrightarrow_{\mathcal{R}} \rho u_n$ gives the desired path. □

Recall that counterexamples of *safety properties* are characterized by finite sequences [4]. Therefore, the above lemma guarantees that $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ does *not* generate spurious counterexamples for safety properties, since any finite counterexample in $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ has a corresponding *real* counterexample in $\bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}$. Together with Corollary 1 and Lemma 3, we have:

Theorem 1. *Given a topmost rewrite theory* $\mathcal{R} = (\Sigma, E, R)$, *and* finite *sets AP and ACT defined by E, for a safety LTLR formula* φ *and a pattern* $t \in N(\mathcal{R})_{AP}$: $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}, [t]_E \models \varphi \iff (\forall \theta : \mathcal{X} \to \mathcal{T}_\Sigma)\ \bar{\mathcal{K}}(\mathcal{R})_{AP,ACT}, [\theta t]_E \models \varphi$.

4 Abstract Narrowing-based LTLR Model Checking

A narrowing-based LKS $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ often has an infinite number of *logical* states (e.g., Figure 2). For narrowing-based LTL model checking, the paper [1] has proposed two abstraction methods to reduce an infinite narrowing-based Kripke structure, namely, *folding abstractions* and *equational abstractions*. This section extends those abstraction techniques to narrowing-based LTLR model checking for trying to reduce an infinite *narrowing-based LKS* to a finite one.

Folding Abstractions. Given a *transition system* $\mathcal{A} = (A, \longrightarrow_{\mathcal{A}})$ with a set of states A and a transition relation $\longrightarrow_{\mathcal{A}} \subseteq A^2$, we can reduce it by collapsing each state a into a *previously seen* state b, while traversing $\mathcal{A}$ from a set of initial states $I \subseteq A$, whenever b is *more general* than a according to a folding relation $a \preccurlyeq b$ [6]. For a set of states $B \subseteq A$, let $Post_{\mathcal{A}}(B) = \{a \in A \mid \exists b \in B.\ b \longrightarrow_{\mathcal{A}} a\}$ (i.e., the *successors* of B) and $Post^*_{\mathcal{A}}(B) = \bigcup_{i \in \mathbb{N}} (Post_{\mathcal{A}})^i(B)$.

Definition 7. *Given* $\mathcal{A} = (A, \longrightarrow_{\mathcal{A}})$ *and a folding relation* $\preccurlyeq \subseteq A^2$, *the* folding abstraction *of* $\mathcal{A}$ *from* $I \subseteq A$ *is* $\mathcal{R}each^{\preccurlyeq}_{\mathcal{A}}(I) = (Post^*_{\mathcal{A}_{\preccurlyeq}}(I), \longrightarrow_{\mathcal{R}each^{\preccurlyeq}_{\mathcal{A}}(I)})$, *where:* $Post^*_{\mathcal{A}_{\preccurlyeq}}(I) = \bigcup_{i \in \mathbb{N}} Post^i_{\mathcal{A}_{\preccurlyeq}}(I)$ *and* $\longrightarrow_{\mathcal{R}each^{\preccurlyeq}_{\mathcal{A}}(I)} = \bigcup_{i \in \mathbb{N}} \longrightarrow^{\preccurlyeq}_{\mathcal{A},i}$ *such that:*

$$
\begin{aligned}
Post^0_{\mathcal{A}_{\preccurlyeq}}(I) &= I, \qquad\qquad \longrightarrow^{\preccurlyeq}_{\mathcal{A},0} = \emptyset,\\
Post^{n+1}_{\mathcal{A}_{\preccurlyeq}}(I) &= \{a \in Post_{\mathcal{A}}(Post^n_{\mathcal{A}_{\preccurlyeq}}(I)) \mid \forall l \leq n\ \forall b \in Post^l_{\mathcal{A}_{\preccurlyeq}}(I).\ a \not\preccurlyeq b\},\\
\longrightarrow^{\preccurlyeq}_{\mathcal{A},n+1} &= \{(a, a') \in Post^n_{\mathcal{A}_{\preccurlyeq}}(I) \times \bigcup_{0 \leq i \leq n+1} Post^i_{\mathcal{A}_{\preccurlyeq}}(I) \mid \exists b \in Post_{\mathcal{A}}(a).\ b \preccurlyeq a'\}.
\end{aligned}
$$

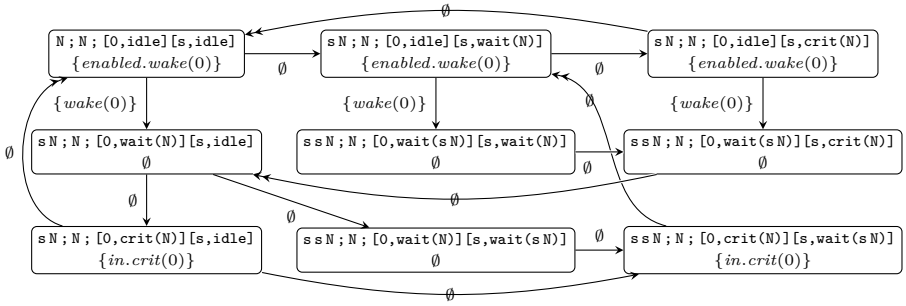


Fig. 3. A folding abstraction for the bakery protocol using the folding relation $\preccurlyeq_E$, where a double-headed arrow denotes a "folded" transition.

For the bakery example, using the E-subsumption $\preccurlyeq_E$ as a folding relation, we have the *finite* folding abstraction $\mathcal{R}each^{\preccurlyeq_E}_{\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}}(\{\texttt{N ; N ; [0,idle][s,idle]}\})$ of $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ from the initial state `N ; N ; [0,idle][s,idle]` in Figure 3.

If a folding relation $\preccurlyeq$ is a total simulation from $\mathcal{A}$ to $\mathcal{A}$, then $\mathcal{R}each^{\preccurlyeq}_{\mathcal{A}}(I)$ simulates the *reachable* subsystem $\mathcal{R}each_{\mathcal{A}}(I) = (Post^*_{\mathcal{A}}(I), \longrightarrow_{\mathcal{A}} \cap\, Post^*_{\mathcal{A}}(I)^2)$ that only contains reachable states from I (i.e., $\preccurlyeq$ is a total simulation from $\mathcal{R}each_{\mathcal{A}}(I)$ to $\mathcal{R}each^{\preccurlyeq}_{\mathcal{A}}(I)$) [1]. Indeed, $\preccurlyeq_E$ for a topmost rewrite theory $\mathcal{R}$ is a total simulation from $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ to $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ (which can be proved in a similar way to Lemma 3). Therefore, $\preccurlyeq_E$ defines a total simulation from $\mathcal{R}each_{\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}}(I)$ to $\mathcal{R}each^{\preccurlyeq_E}_{\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}}(I)$. Consequently, by Corollary 1:

Theorem 2. *For an LTLR formula φ and a pattern $t \in N(\mathcal{R})_{AP}$, we have that $\mathcal{R}each^{\preccurlyeq_E}_{\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}}(\{[t]_E\}), [t]_E \models \varphi$ implies $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}, [t]_E \models \varphi$.*

For the bakery example, the liveness property $\Diamond \mathit{in.crit}(0)$ under the fairness assumption $\Diamond\Box \mathit{enabled.wake}(0) \rightarrow \Box\Diamond \mathit{wake}(0)$ holds in the folding abstraction $\mathcal{R}each^{\preccurlyeq_E}_{\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}}(\{\texttt{N ; N ; [0,idle][s,idle]}\})$ of Figure 3, because any infinite paths continuously staying in the first row violate the fairness assumption. Hence, this property is also satisfied for any related concrete system.

Equational Abstractions. In general, a folding abstraction of a narrowing-based LKS is *not* finite. For the bakery example, there exists an infinite path within the folding abstraction from `N ; N ; [0,idle] IS` in Figure 4, which keeps incrementing the number of processes with instantiations. To further reduce an infinite logical state space, we can apply equational abstractions to eventually obtain a finite abstract narrowing-based LKS for LTLR model checking.

Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, by adding a set of equations G such that $\mathit{true} \neq_{E \cup G} \mathit{false}$, we define an *equational abstraction* $\mathcal{R}/G = (\Sigma, E \cup G, R)$ [15]. It specifies the quotient abstraction $\bar{\mathcal{N}}(\mathcal{R}/G)_{AP,ACT}$ by the equivalence relation $\equiv_G$ on states, namely, $[t]_E \equiv_G [t']_E$ iff $t =_{E \cup G} t'$. Provided that a set of

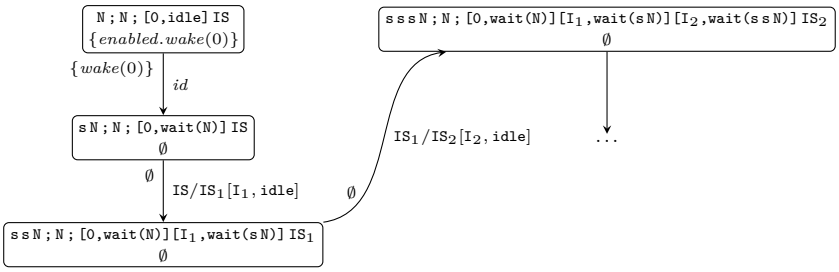


Fig. 4. An infinite path in the folding abstraction for the bakery protocol with an unbounded number of processes, where `IS` stands for a set of *idle* processes.

state propositions AP and a set of spatial action patterns ACT are defined by E, the condition $true \neq_{E \cup G} false$ ensures that any two states with $t =_{E \cup G} t'$ satisfy the same set of state propositions. Similarly, any two one-step proof terms with $l(\sigma_l) =_{E \cup G} l'(\sigma_{l'})$ satisfy the same set of spatial action patterns.

Similar to the cases of LTL model checking [1,15], an equational abstraction $\bar{\mathcal{N}}(\mathcal{R}/G)_{AP,ACT}$ simulates the narrowing-based LKS $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$.

Lemma 5. *Given a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$, finite sets AP and ACT defined by E, and a set G of equations, if $true \neq_{E \cup G} false$, then there exists a total simulation from $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ to $\bar{\mathcal{N}}(\mathcal{R}/G)_{AP,ACT}$.*

Proof. Let $H_G = \{([t]_E, [t]_{E \cup G}) \mid t \in N(\mathcal{R})_{AP}\}$. Suppose that $[t]_E \xrightarrow{A}_{\bar{\mathcal{N}}(\mathcal{R})} [t']_E$ and $t =_{E \cup G} u$. By definition, there are σ and ζ such that $t \rightsquigarrow_{l,\sigma,\mathcal{R}} t''$ by a rule $l : q \longrightarrow r \in R$ and $t' = \zeta t''$, where $\sigma \in CSU_E(t = q)$, $t'' = \sigma r$, and $\zeta \in CSU_E(\bigwedge_{1 \leq i \leq n}(t'' \models p_i) = b_i \ \wedge \ \bigwedge_{1 \leq j \leq m}(l(\sigma_l) \models \delta_j) = b_{n+j})$ for some $b_1, \ldots, b_{n+m} \in \{true, false\}$, given $AP = \{p_1, \ldots, p_n\}$ and $ACT = \{\delta_1, \ldots, \delta_m\}$. Since $\sigma \in CSU_E(t = q)$, $\exists \sigma' \in CSU_{E \cup G}(u = q)$ such that $\sigma =_{E \cup G} \sigma'$. Then, $u \rightsquigarrow_{l,\sigma',\mathcal{R}/G} u'$ using the same rule $l : q \longrightarrow r$, where $u' = \sigma' r =_{E \cup G} \sigma r = t''$. Notice that $(t'' \models p_i) =_{E \cup G} (u' \models p_i)$ and $(l(\sigma_l) \models \delta_j) =_{E \cup G} (l(\sigma'_l) \models \delta_j)$. Thus, $\exists \zeta' \in CSU_{E \cup G}(\bigwedge_{1 \leq i \leq n}(u' \models p_i) = b_i \ \wedge \ \bigwedge_{1 \leq j \leq m}(l(\sigma'_l) \models \delta_j) = b_{n+j})$ with $\zeta =_{E \cup G} \zeta'$. Thus, $[u]_{E \cup G} \xrightarrow{A}_{\bar{\mathcal{N}}(\mathcal{R}/G)} [\zeta' u']_{E \cup G}$, where $\zeta' u' =_{E \cup G} \zeta t'' = t'$. Since $true \neq_{E \cup G} false$, $[t']_E$ and $[\zeta' u']_{E \cup G}$ satisfy the same state propositions. Therefore, H_G is a total simulation from $\bar{\mathcal{N}}(\mathcal{R})_{AP,ACT}$ to $\bar{\mathcal{N}}(\mathcal{R}/G)_{AP,ACT}$. □

For the bakery example, by adding the following equations that collapses extra waiting processes with non-zero identifiers, where `ICPS` denotes a set of *idle* or *crit* processes, and `WP3` denotes *zero or at most three wait* processes:

```
eq [NZ,D] = [D] .                          -- remove non-zero identifiers
eq s s s N M ; M ; ICPS WP3 [wait(s N M)] [wait(s s N M)]
 =   s s N M ; M ; ICPS WP3 [wait(s N M)] .
```

we have the folded abstract narrowing-based LKS in Figure 5, provided with the extra spatial action pattern *wake* that holds if the *wake* rule is applied.

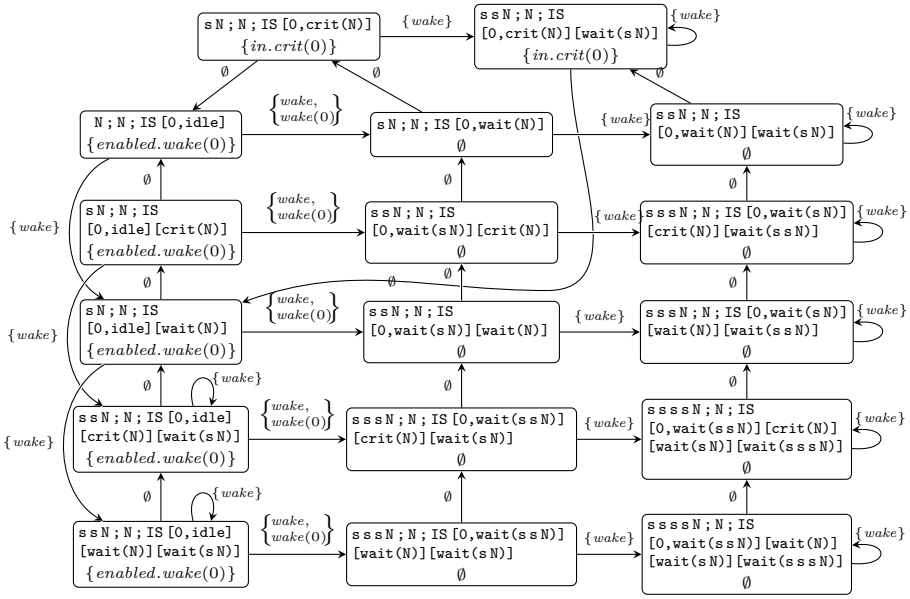


Fig. 5. An folded equational abstraction for the bakery protocol.

We can easily see that there is a counterexample of the property $\Diamond in.crit(0)$ under $\Diamond\Box enabled.wake(0) \rightarrow \Box\Diamond wake(0)$ in which the *wake* rule is continuously applied forever, which is impossible if there is a finite number of processes. Assuming the extra fairness assumption $\Box\Diamond\neg wake$, the property $\Diamond in.crit(0)$ is now satisfied since any infinite paths staying in the first column forever violate $\Diamond\Box enabled.wake(0) \rightarrow \Box\Diamond wake(0)$, and any paths staying in a self loop forever violate $\Box\Diamond\neg wake$. Consequently, under the fairness assumptions, $\Diamond in.crit(0)$ is satisfied for an unbounded number of processes.

5 Related Work and Conclusions

A number of infinite-state model checking methods have been developed based on symbolic and abstraction techniques; see [1,6] for an overview and comparison with narrowing-based model checking. To the best of our knowledge, our work proposes the first *symbolic* model checking method to verify LTLR properties of infinite-state systems. For finite-state systems the paper [2] presents various model checking algorithms for LTLR properties. LTLR is a sublogic of TLR^* that generalizes the state-based logic CTL^* (see [14] for related work). On the topic of *complement patterns*, the most closely related work is [8,9,12]. We plan to use their ideas, as well as ongoing work by Skeirik and Meseguer on the concept of *B-linear terms* in order-sorted signatures, which are pattern terms whose syntactic structure guarantees the existence of complements modulo B, to automate the full equational definition of satisfaction of spatial action patterns.

In conclusion, this work should be understood as a contribution that increases the expressive power of infinite-state model checking methods. Specifically, the expressive power of narrowing-based infinite-state model checking has been extended form LTL to LTLR, allowing temporal properties that can use both state predicates and action patterns. This extension is nontrivial because of the need for building a symbolic transition system where states are AP-instantiated and transitions are ACT-instantiated. All the necessary theoretical foundations are now in place for embarking into a future implementation of a narrowing-based LTLR model checker in Maude in the spirit of the similar LTL tool described in [1]. As done in [1], for the LTLR tool we will be able to rely on the extensive body of work on efficient LTLR model checking algorithms described in [2]. Beyond these goals, the integration of constraints and SMT solving within the planned narrowing-based LTLR model checker, as well as the study of more flexible "stuttering" AP/ACT-simulations, are also exciting possibilities.

References

1. Bae, K., Escobar, S., Meseguer, J.: Abstract logical model checking of infinite-state systems using narrowing. In: RTA. LIPIcs, vol. 21, pp. 81–96 (2013)
2. Bae, K., Meseguer, J.: Model checking linear temporal logic of rewriting formulas under localized fairness. Science of Computer Programming (2014), to appear
3. Chaki, S., Clarke, E., Ouaknine, J., Sharygina, N., Sinha, N.: State/event-based software model checking. In: IFM. LNCS, vol. 2999, pp. 128–147. Springer (2004)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2001)
5. Comon-Lundh, H., Delaune, S.: The finite variant property: How to get rid of some algebraic properties. Term Rewriting and Applications pp. 294–307 (2005)
6. Escobar, S., Meseguer, J.: Symbolic model checking of infinite-state systems using narrowing. In: RTA. pp. 153–168 (2007)
7. Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. J. Algebraic and Logic Programming 81, 898–928 (2012)
8. Fernández, M.: AC complement problems: Satisfiability and negation elimination. J. Symb. Comput. 22(1), 49–82 (1996)
9. Fernández, M.: Negation elimination in empty or permutative theories. J. Symb. Comput. 26(1), 97–133 (1998)
10. Hullot, J.M.: Canonical forms and unification. In: CADE. LNCS vol. 87, Springer (1980)
11. Jouannaud, J.P., Kirchner, C., Kirchner, H.: Incremental construction of unification algorithms in equational theories. In: ICALP. LNCS, vol. 154. Springer (1983)
12. Lassez, J.L., Marriott, K.: Explicit representation of terms defined by counter examples. J. Autom. Reasoning 3(3), 301–317 (1987)
13. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
14. Meseguer, J.: The temporal logic of rewriting: A gentle introduction. In: Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 354–382. Springer (2008)
15. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theoretical Computer Science 403(2-3), 239–264 (2008)
16. Meseguer, J., Thati, P.: Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. Higher-Order and Symbolic Computation 20(1–2), 123–160 (2007)

Modelling and verifying contract-oriented systems in Maude

Massimo Bartoletti[1], Maurizio Murgia[1], Alceste Scalas[1], and Roberto Zunino[2]

[1] Università degli Studi di Cagliari, Italy
[2] Università degli Studi di Trento, Italy

Abstract. We address the problem of modelling and verifying contract-oriented systems, wherein distributed agents may advertise and stipulate contracts, but — differently from most other approaches to distributed agents — are not assumed to always behave "honestly". We describe an executable specification in Maude of the semantics of CO_2, a calculus for contract-oriented systems [6]. The *honesty* property [5] characterises those agents which always respect their contracts, in *all* possible execution contexts. Since there is an infinite number of such contexts, honesty cannot be directly verified by model-checking the state space of an agent (indeed, honesty is an undecidable property in general [5]). The main contribution of this paper is a sound verification technique for honesty. To do that, we safely over-approximate the honesty property by abstracting from the actual contexts a process may be engaged with. Then, we develop a model-checking technique for this abstraction, we describe an implementation in Maude, and we discuss some experiments with it.

1 Introduction

Contract-oriented computing is a software design paradigm where the interaction between clients and services is disciplined through contracts [6,4]. Contract-oriented services start their life-cycle by advertising contracts which specify their required and offered behaviour. When compliant contracts are found, a session is created among the respective services, which may then start interacting to fulfil their contracts. Differently from other design paradigms (e.g. those based on the session types discipline [10]), services are not assumed to be *honest*, in that they might not respect the promises made [5]. This may happen either unintentionally (because of errors in the service specification), or because of malicious behaviour.

Dishonest behaviour is assumed to be automatically detected and sanctioned by the service infrastructure. This gives rise to a new kind of attacks, that exploit possible discrepancies between the promised and the actual behaviour. If a service does not behave as promised, an attacker can induce it to a situation where the service is sanctioned, while the attacker is reckoned honest. A crucial problem is then how to avoid that a service results definitively culpable of a contract violation, despite of the honest intentions of its developer.

In this paper we present an executable specification in Maude [9] of CO_2, a calculus for contract-oriented computing [4]. Furthermore, we devise and implement a sound verification technique for honesty. We start in § 2 by introducing a

new model for contracts. Borrowing from other approaches to behavioural contracts [8,5], ours are bilateral contracts featuring internal/external choices, and recursion. We define and implement in Maude two crucial primives on contracts, i.e. *compliance* and *culpability testing*, and we study some relevant properties.

In § 3 we present CO_2 (instantiated with the contracts above), and an executable specification of its semantics in Maude. In § 4 we formalise a weak notion of honesty, i.e. when a process P is honest *in a given context*, and we implement and experiment with it through the Maude model checker.

The main technical results follow in § 5, where we deal with the problem of checking honesty in *all* possible contexts. To do that, we start by defining an abstract semantics of CO_2, which preserves the transitions of a participant $\mathsf{A}[P]$, while abstracting those of the context wherein $\mathsf{A}[P]$ is run. Building upon the abstract semantics, we then devise an abstract notion of honesty (α-*honesty*, Def. 11), which neglects the execution context. Theorem 5 states that α-honesty correctly approximates honesty, and that — under certain hypotheses — it is also complete. We then propose a verification technique for α-honesty, and we provide an implementation in Maude. Some experiments have then been carried out; quite notably, our tool has allowed us to determine the dishonesty of a supposedly-honest CO_2 process appeared in [5] (see Ex. 5).

Because of space limits, we make available online the proofs of all our statements, as well as the Maude implementation, and the experiments made [2].

2 Modelling contracts

We model contracts as processes in a simple algebra, with internal/external choice and recursion. Compliance between contracts ensures progress, until a successful state is reached. We prove that our model enjoys some relevant properties. First, in each non-final state of a contract there is exactly one participant who is *culpable*, i.e., expected to make the next move (Theorem 1). Furthermore, a participant always recovers from culpability in at most two steps (Theorem 2).

Syntax. We assume a finite set of *participant names* (ranged over by $\mathsf{A}, \mathsf{B}, \ldots$) and a denumerable set of *atoms* (ranged over by $\mathsf{a}, \mathsf{b}, \ldots$). We postulate an involution $\mathrm{co}(\mathsf{a})$, also written as $\bar{\mathsf{a}}$, extended to sets of atoms in the natural way. Def. 1 introduces the syntax of contracts. We distinguish between (*unilateral*) contracts c, which model the promised behaviour of a single participant, and *bilateral* contracts γ, which combine the contracts advertised by two participants.

Definition 1. Unilateral contracts *are defined by the following grammar:*

$$c, d \;::=\; \textstyle\bigoplus_{i\in\mathcal{I}} \mathsf{a}_i \,;\, c_i \;\;\big|\;\; \sum_{i\in\mathcal{I}} \mathsf{a}_i \,.\, c_i \;\;\big|\;\; \mathit{ready}\ \mathsf{a}.c \;\;\big|\;\; \mathrm{rec}\ X.\, c \;\big|\; X$$

where (*i*) *the index set* $\mathcal{I}$ *is finite;* (*ii*) *the "ready" prefix may appear at the top-level, only;* (*iii*) *recursion is guarded.*

Bilateral contracts γ *are terms of the form* A *says* $c \mid \mathsf{B}$ *says* d, *where* $\mathsf{A} \neq \mathsf{B}$ *and at most one occurrence of "ready" is present. The order of unilateral contracts in* γ *is immaterial, i.e.* A *says* $c \mid \mathsf{B}$ *says* $d \;\equiv\; \mathsf{B}$ *says* $d \mid \mathsf{A}$ *says* c.

An internal sum $\bigoplus_{i\in\mathcal{I}} \mathsf{a}_i\,;\, c_i$ allows to choose one of the branches $\mathsf{a}_i\,;\, c_i$, to perform the action a_i, and then to behave according to c_i. Dually, an external sum $\sum_{i\in\mathcal{I}} \mathsf{a}_i\,.\, c_i$ allows to wait for the other participant to choose one of the branches $\mathsf{a}_i\,.\, c_i$, then to perform the corresponding a_i and behave according to c_i. Separators ; and . allow for distinguishing singleton internal sums $\mathsf{a}\,;\, c$ from singleton external sums $\mathsf{a}\,.\, c$. Empty internal/external sums are denoted with 0. We will only consider contracts without free occurrences of recursion variables X.

Example 1. An online store A has the following contract: buyers can iteratively add items to the shopping cart (`addToCart`); when at least one item has been added, the client can either `cancel` the order or `pay`; then, the store can accept (`ok`) or decline (`no`) the payment. Such a contract may be expressed as c_{A} below:

$$\begin{aligned} c_{\text{pay}} &= \mathsf{pay}\,.\,\big(\overline{\mathsf{ok}}\,;\, 0 \,\oplus\, \overline{\mathsf{no}}\,;\, 0\big) \\ c_{\mathsf{A}} &= \mathsf{addToCart}\,.\,(\text{rec } Z.\, \mathsf{addToCart}\,.\, Z \;+\; c_{\text{pay}} \;+\; \mathsf{cancel}\,.\, 0) \end{aligned}$$

Instead, a buyer contract could be expressed as:

$$c_{\mathsf{B}} \;=\; \text{rec } Z.\, \big(\overline{\mathsf{addToCart}}\,;\, Z \;\oplus\; \overline{\mathsf{pay}}\,;\, (\mathsf{ok}\,.\, 0 \;+\; \mathsf{no}\,.\, 0)\big)$$

The Maude specification of the syntax of contracts is defined as follows:

```
sorts Atom UniContract Participant AdvContract BiContract
      IGuarded EGuarded IChoice EChoice Var Id RdyContract .
subsort Id < IGuarded < IChoice < UniContract < RdyContract .
subsort Id < EGuarded < EChoice < UniContract < RdyContract .
subsort Var < UniContract .
```

The sorts `IGuarded` and `EGuarded` represent singleton internal/external sums, respectively, while `IChoice` and `EChoice` are for arbitrary internal/external sums. `Id` represents empty sums, and it is a subsort of internal and external sums (either singleton or not). `RdyContract` if for contracts which may have a top-level *ready* , while `AdvContract` is a unilateral contract advertised by some participant.

```
op -_ : Atom -> Atom [ctor] .
eq - - a:Atom = a:Atom .
op 0 : -> Id [ctor] .
op _._ : Atom UniContract -> EGuarded [frozen ctor] .
op _;_ : Atom UniContract -> IGuarded [frozen ctor] .
op _+_ : EChoice EChoice -> EChoice [frozen comm assoc id: 0 ctor] .
op _(+)_ : IChoice IChoice -> IChoice [frozen comm assoc id: 0 ctor] .
op ready _._ : Atom UniContract -> RdyContract [frozen ctor] .
op rec _._ : Var IChoice -> UniContract [frozen ctor] .
op rec _._ : Var EChoice -> UniContract [frozen ctor] .
op _ says _  : Participant RdyContract -> AdvContract [ctor] .
op _ | _ : AdvContract AdvContract -> BiContract [comm ctor] .
```

The operator - models the involution on atoms, with `eq - - a:Atom = a:Atom`. The other operators are rather standard, and they guarantee that each `UniContract` respects the syntactic constraints imposed by Def. 1.

Semantics. The evolution of bilateral contracts is modelled by $\xrightarrow{\mu}\!\!\!\!\rightarrow$, the smallest relation closed under the rules in Fig. 1 and under $\equiv$. The congruence $\equiv$ is

$$\mathsf{A}\ \mathit{says}\ (\mathsf{a}\,;\, c \oplus c') \mid \mathsf{B}\ \mathit{says}\ (\bar{\mathsf{a}}\,.\,d + d') \xrightarrow{\mathsf{A}\ \mathit{says}\ \mathsf{a}} \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says\ ready}\ \bar{\mathsf{a}}.d \quad [\textsc{IntExt}]$$

$$\mathsf{A}\ \mathit{says\ ready}\ \mathsf{a}.\ c \mid \mathsf{B}\ \mathit{says}\ d \xrightarrow{\mathsf{A}\ \mathit{says}\ \mathsf{a}} \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d \quad [\textsc{Rdy}]$$

Fig. 1. Semantics of contracts (symmetric rules for B actions omitted)

the least relation including α-conversion of recursion variables, and satisfying $\mathrm{rec}\ X.\ c \equiv c\{^{\mathrm{rec}\ X.\ c}/_X\}$ and $\bigoplus_{i\in\emptyset} \mathsf{a}_i\,;\, c_i \equiv \sum_{i\in\emptyset} \mathsf{a}_i\,.\,c_i$. The label $\mu = \mathsf{A}\ \mathit{says}\ \mathsf{a}$ models A performing action a. Hereafter, we shall consider contracts up-to $\equiv$.

In rule [IntExt], participant A selects the branch a in an internal sum, and B is then forced to commit to the corresponding branch $\bar{\mathsf{a}}$ in his external sum. This is done by marking that branch with *ready* $\bar{\mathsf{a}}$, while discarding all the other branches; B will then perform his action in the subsequent step, by rule [Rdy].

In Maude, the semantics of contracts is an almost literal translation of that in Fig. 1 (except that labels are moved to configurations). The one-step transition relation is defined as follows:

```
crl [IntExt]: A says a ; c (+) c' | B says b . d + d'
 => {A says a} A says c | B says ready b . d                if a = - b .

rl [Rdy]: A says ready a.c | B says d => {A says a} A says c | B says d .
```

Compliance. Two contracts are *compliant* if, whenever a participant A wants to choose a branch in an internal sum, then participant B always offers A the opportunity to do it. To formalise compliance, we first define a partial function rdy from bilateral contracts to sets of atoms. Intuitively, if the unilateral contracts in γ do not agree on the first step, then $\mathrm{rdy}(\gamma)$ is undefined (i.e. equal to $\bot$). Otherwise, $\mathrm{rdy}(\gamma)$ contains the atoms which could be fired in the first step.

Definition 2 (Compliance). *Let the partial function* rdy *be defined as:*

$$\mathrm{rdy}\left(\mathsf{A}\ \mathit{says} \bigoplus_{i\in\mathcal{I}} \mathsf{a}_i\,;\, c_i \mid \mathsf{B}\ \mathit{says} \sum_{j\in\mathcal{J}} \mathsf{b}_j\,.\,c_j\right) = \{\mathsf{a}_i\}_{i\in\mathcal{I}} \quad \begin{array}{l} \mathit{if} \quad \{\mathsf{a}_i\}_{i\in\mathcal{I}} \subseteq \{\bar{\mathsf{b}}_j\}_{j\in\mathcal{J}} \\ \mathit{and}\ \ (\mathcal{I}=\emptyset \implies \mathcal{J}=\emptyset) \end{array}$$

$$\mathrm{rdy}(\mathsf{A}\ \mathit{says\ ready}\ \mathsf{a}.c \mid \mathsf{B}\ \mathit{says}\ d) = \{\mathsf{a}\}$$

Then, the compliance relation $\bowtie$ between unilateral contracts is the largest relation such that, whenever $c \bowtie d$:

(1) $\mathrm{rdy}(\mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d) \neq \bot$
(2) $\mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d \xrightarrow{\mu}\!\!\!\!\rightarrow \mathsf{A}\ \mathit{says}\ c' \mid \mathsf{B}\ \mathit{says}\ d' \implies c' \bowtie d'$

Example 2. Let $\gamma = \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d$, where $c = \mathsf{a}\,;\, c_1 \oplus \mathsf{b}\,;\, c_2$ and $d = \bar{\mathsf{a}}\,.\,d_1 + \bar{\mathsf{c}}\,.\,d_2$. If the participant A internally chooses to perform a, then γ will take a transition to $\mathsf{A}\ \mathit{says}\ c_1 \mid \mathsf{B}\ \mathit{says\ ready}\ \bar{\mathsf{a}}.d_1$. Suppose instead that A chooses to perform b, which is not offered by B in his external choice. In this case, $\gamma \not\xrightarrow{\mathsf{A}\ \mathit{says}\ \mathsf{b}}\!\!\!\!\rightarrow$. We have that $\mathrm{rdy}(\gamma) = \bot$, which does not respect item (1) of Def. 2. Therefore, c and d are *not* compliant.

We say that a contract is *proper* if the prefixes of each summation are pairwise distinct. The next lemma states that each proper contract has a compliant one.

Lemma 1. *For all proper contracts c, there exists d such that $c \bowtie d$.*

Def. 2 cannot be directly exploited as an algorithm for checking compliance. Lemma 2 gives an alternative, model-checkable characterisation of $\bowtie$.

Lemma 2. *For all bilateral contracts $\gamma = \mathsf{A}$ says $c \mid \mathsf{B}$ says d:*

$$c \bowtie d \iff (\forall \gamma'.\ \gamma \twoheadrightarrow^* \gamma' \implies \mathrm{rdy}(\gamma') \neq \bot)$$

In Maude, the compliance relation is defined as suggested by Lemma 2. The predicate `isBottom` is true for a contract γ whenever $\mathrm{rdy}(\gamma) = \bot$. The operator `<>` used below allows for the transitive closure of the transition relation. The relation `c |X| d` is implemented by verifying that the contract A *says* $c \mid$ B *says* d satisfies the LTL formula $\Box\neg$ `isBottom`. This is done through the Maude model checker.

```
eq <{l} g> |= isBottom = is rdy(g) eq bottom .
op _|X|_ : UniContract UniContract -> Bool .
eq c |X| d = modelCheck(<A says c | B says d>, [] ~isBottom) == true .
```

Example 3. Recall the store contract c_{A} in Ex. 1. Its Maude version is:

```
op Z : -> Var .
ops addToCart pay ok no cancel : -> Atom .
ops CA CPay CB : -> UniContract .
eq CPay = pay . (- ok ; 0 (+) - no ; 0) .
eq CA = addToCart . (rec Z . addToCart . Z + CPay + cancel . 0) .
```

Instead, the Maude implementation of the buyer contract c_{B} in Ex. 1 is:

```
eq CB = rec Z . ( - addToCart ; Z (+) - pay ; (ok . 0 + no . 0)) .
```

We can verify with Maude that `CA` and `CB` are *not* compliant:

```
red CA |X| CB .
result Bool: false
```

The problem is that `CB` may choose to `pay` even when the cart is empty. We can easily fix the buyer contract as follows, and then obtain compliance:

```
red CA |X| (- addToCart ; CB) .
result Bool: true
```

Culpability. We now tackle the problem of determining who is expected to make the next step for the fulfilment of a bilateral contract. We call a participant A *culpable* in γ if she is expected to perform some actions so to make γ progress.

Definition 3. *A participant* A *is culpable in γ ($\mathsf{A} \dot{\frown} \gamma$ in symbols) iff $\gamma \xrightarrow{\mathsf{A}\ says\ \mathtt{a}} \!\!\!\!\twoheadrightarrow$ for some* `a`*. When* A *is* not *culpable in γ we write $\mathsf{A} \dot{\smile} \gamma$.*

Theorem 1 below establishes that, when starting with compliant contracts, exactly one participant is culpable in a bilateral contract. The only exception is A *says* $0 \mid$ B *says* 0, which represents a successfully terminated interaction, where nobody is culpable.

Theorem 1. *Let* $\gamma = \mathsf{A}$ *says* $c \mid \mathsf{B}$ *says* d*, with* $c \bowtie d$*. If* $\gamma \twoheadrightarrow^* \gamma'$*, then either* $\gamma' = \mathsf{A}$ *says* $0 \mid \mathsf{B}$ *says* 0*, or there exists a unique culpable in* γ'*.*

The following theorem states that a participant is always able to recover from culpability by performing some of her duties. This requires at most two steps.

Theorem 2 (Contractual exculpation). *Let* $\gamma = \mathsf{A}$ *says* $c \mid \mathsf{B}$ *says* d*. For all* γ' *such that* $\gamma \twoheadrightarrow^* \gamma'$*, we have that:*

(1) $\gamma' \not\twoheadrightarrow \implies \mathsf{A} \breve{\cup} \gamma'$ *and* $\mathsf{B} \breve{\cup} \gamma'$

(2) $\mathsf{A} \breve{\cap} \gamma' \implies \forall \gamma''.\gamma' \twoheadrightarrow \gamma'' \implies \begin{cases} \mathsf{A} \breve{\cup} \gamma'', \text{ or} \\ \forall \gamma'''.\gamma'' \twoheadrightarrow \gamma''' \implies \mathsf{A} \breve{\cup} \gamma''' \end{cases}$

Item (1) of Theorem 2 says that, in a stuck contract, no participant is culpable. Item (2) says that if A is culpable, then she can always exculpate herself in *at most* two steps, i.e.: one step if A has an internal choice, or a *ready* followed by an external choice; two steps if A has a *ready* followed by an internal choice.

We specify culpability in Maude as follows. The formula `{l} g |= --A-->>` is true whenever `g` has been reached by some transitions of `A`. The participant `A` is culpable in `g`, written `A :C g`, if `g` satisfies the LTL formula `O --A-->>` (where `O` is the "next" operator of LTL). This is verified through the Maude model checker.

```
op --_->> : Participant -> Prop .
eq {A says a} g |= -- A ->> = true .
eq {l} g |= -- A ->> = false [owise] .
op _ :C _ : Participant BiContract -> Bool .
eq A :C g = modelCheck(g, O -- A ->>) == true .
```

3 Modelling contracting processes

We model agents and systems through the process calculus CO_2 [3], which we instantiate with the contracts introduced in § 2. The primitives of CO_2 allow agents to advertise contracts, to open sessions between agents with compliant contracts, to execute them by performing some actions, and to query contracts.

Syntax. Let $\mathcal{V}$ and $\mathcal{N}$ be disjoint sets of *session variables* (ranged over by $x, y, \ldots$) and *session names* (ranged over by $s, t, \ldots$). Let $u, v, \ldots$ range over $\mathcal{V} \cup \mathcal{N}$, and $\boldsymbol{u}, \boldsymbol{v}$ range over $2^{\mathcal{V} \cup \mathcal{N}}$.

Definition 4. *The syntax of CO₂ is given as follows:*

Systems	S	$::=$	$\mathbf{0}$	$\mid$	$\mathsf{A}[P]$	$\mid$	$s[\gamma]$	$\mid$ $S \mid S$	$\mid$ $(u)S$	$\mid$ $\{\downarrow_u c\}_{\mathsf{A}}$
Processes	P	$::=$	$\sum_i \pi_i.P_i$	$\mid$	$P \mid P$	$\mid$	$(u)P$	$\mid$ $X(\boldsymbol{u})$		
Prefixes	π	$::=$	τ	$\mid$	$\mathsf{tell} \downarrow_u c$	$\mid$	$\mathsf{do}_u\, \mathsf{a}$	$\mid$ $\mathsf{ask}_u \phi$		

commutative monoidal laws for $|$ on processes and systems

$$\mathsf{A}[(v)P] \equiv (v)\,\mathsf{A}[P] \qquad Z \mid (u)Z' \equiv (u)(Z \mid Z') \;\text{ if } u \notin \mathrm{fv}(Z) \cup \mathrm{fn}(Z)$$

$$(u)(v)Z \equiv (v)(u)Z \qquad (u)Z \equiv Z \;\text{ if } u \notin \mathrm{fv}(Z) \cup \mathrm{fn}(Z) \qquad \{\downarrow_s c\}_{\mathsf{A}} \equiv \mathbf{0}$$

Fig. 2. Structural equivalence for CO_2 (Z, Z' range over systems or processes).

Systems are the parallel composition of *participants* $\mathsf{A}[P]$, *delimited systems* $(u)S$, *sessions* $s[\gamma]$ and *latent contracts* $\{\downarrow_u c\}_{\mathsf{A}}$. A latent contract $\{\downarrow_x c\}_{\mathsf{A}}$ represents a contract c (advertised by A) which has not been stipulated yet; upon stipulation, the variable x will be instantiated to a fresh session name. We assume that, in a system of the form $(\boldsymbol{u})(\mathsf{A}[P] \mid \mathsf{B}[Q]) \mid \cdots)$, $\mathsf{A} \neq \mathsf{B}$. We denote with K a special participant name (playing the role of contract broker) such that, in each system $(\boldsymbol{u})(\mathsf{A}[P] \mid \cdots)$, $\mathsf{A} \neq \mathsf{K}$. We allow for prefix-guarded finite sums of processes, and write $\pi_1.P_1 + \pi_2.P_2$ for $\sum_{i\in\{1,2\}} \pi_i.P_i$, and $\mathbf{0}$ for $\sum_{\emptyset} P$. Recursion is allowed only for processes; we stipulate that each process identifier X has a unique defining equation $X(x_1, \ldots, x_j) \stackrel{\text{def}}{=} P$ such that $\mathrm{fv}(P) \subseteq \{x_1, \ldots, x_j\} \subseteq \mathcal{V}$, and each occurrence of process identifiers in P is prefix-guarded. We will sometimes omit the arguments of $X(\boldsymbol{u})$ when they are clear from the context.

Prefixes include silent action τ, contract advertisement $\mathsf{tell} \downarrow_u c$, action execution $\mathsf{do}_u\,\mathsf{a}$, and contract query $\mathsf{ask}_u\,\phi$ (where ϕ is an LTL formula on γ). In each prefix $\pi \neq \tau$, u refers to the target session involved in the execution of π.

In Maude, we translate the syntax of CO_2 almost literally. Here we just show the sorts used; see [2] for the full details.

```
sorts System Process Prefix SessionName SessionVariable SessionIde
      GuardProc Sum IdeVec ProcIde ParamList .
subsort SessionName < SessionIde < IdeVec .
subsort Qid < SessionVariable < SessionIde < IdeVec .
subsort GuardProc < Sum < Process .
subsort SessionIde < ParamList .
```

The sort `SessionIde` is a super sort of both `SessionVariable` and `SessionName`. Session variables can be of sort `Qid`; session names can not. Sort `IdeVec` models sets of `SessionIde` (used as syntactic sugar for delimitations), while `ParamList` models vectors of `SessionIde` (used for parameters of defining equations).

Semantics. The CO_2 semantics is formalised by the relation $\xrightarrow{\mu}$ in Fig. 3, where $\mu \in \{\mathsf{A}:\pi \mid \mathsf{A}\neq\mathsf{K}\} \cup \{\mathsf{K}:\mathsf{fuse}\}$. We will consider processes and systems up-to the congruence relation $\equiv$ in Fig. 2. The axioms for $\equiv$ are fairly standard — except the last one: it collects garbage terms possibly arising from variable substitutions.

Rule [Tau] just fires a τ prefix. Rule [Tell] advertises a latent contract $\{\downarrow_x c\}_{\mathsf{A}}$. Rule [Fuse] finds *agreements* among the latent contracts: it happens when there exist $\{\downarrow_x c\}_{\mathsf{A}}$ and $\{\downarrow_y d\}_{\mathsf{B}}$ such that $\mathsf{A} \neq \mathsf{B}$ and $c \bowtie d$. Once the agreement is reached, a fresh session containing $\gamma = \mathsf{A}$ *says* $c \mid \mathsf{B}$ *says* d is created. Rule [Do]

$$\mathsf{A}[\tau.P + P' \mid Q] \xrightarrow{\mathsf{A}:\ \tau} \mathsf{A}[P \mid Q] \qquad [\textsc{Tau}]$$

$$\mathsf{A}[\mathsf{tell} \downarrow_u c.P + P' \mid Q] \xrightarrow{\mathsf{A}:\ \mathsf{tell} \downarrow_u c} \mathsf{A}[P \mid Q] \ \mid\ \{\downarrow_u c\}_{\mathsf{A}} \qquad [\textsc{Tell}]$$

$$\frac{c \bowtie d \qquad \gamma = \mathsf{A}\ says\ c \mid \mathsf{B}\ says\ d \qquad \sigma = \{^{s}/_{x,y}\} \qquad s \text{ fresh}}{(x,y)(S \ \mid\ \{\downarrow_x c\}_{\mathsf{A}} \ \mid\ \{\downarrow_y d\}_{\mathsf{B}}) \xrightarrow{\mathsf{K}:\ \mathsf{fuse}} (s)(S\sigma \ \mid\ s[\gamma])} \qquad [\textsc{Fuse}]$$

$$\frac{\gamma \xrightarrow{\mathsf{A}\ says\ \mathsf{a}} \gamma'}{\mathsf{A}[\mathsf{do}_s\ \mathsf{a}.P + P' \mid Q] \ \mid\ s[\gamma] \xrightarrow{\mathsf{A}:\ \mathsf{do}_s\ \mathsf{a}} \mathsf{A}[P \mid Q] \ \mid\ s[\gamma']} \qquad [\textsc{Do}]$$

$$\frac{\gamma \vdash \phi}{\mathsf{A}[\mathsf{ask}_s\ \phi.P + P' \mid Q] \ \mid\ s[\gamma] \xrightarrow{\mathsf{A}:\ \mathsf{ask}_s\ \phi} \mathsf{A}[P \mid Q] \ \mid\ s[\gamma]} \qquad [\textsc{Ask}]$$

$$\frac{X(\boldsymbol{u}) \stackrel{\text{def}}{=} P \qquad \mathsf{A}[P\{\boldsymbol{v}/\boldsymbol{u}\} \mid Q] \mid S \xrightarrow{\mu} S'}{\mathsf{A}[X(\boldsymbol{v}) \mid Q] \mid S \xrightarrow{\mu} S'} \ [\textsc{Def}] \qquad \frac{S \xrightarrow{\mu} S'}{S \mid S'' \xrightarrow{\mu} S' \mid S''} \ [\textsc{Par}]$$

$$\frac{S \xrightarrow{\mathsf{A}:\ \pi} S'}{(u)S \xrightarrow{\mathsf{A}:\ \mathrm{del}_u(\pi)} (u)S'} \ [\textsc{Del}] \qquad \text{where } \mathrm{del}_u(\pi) = \begin{cases} \tau & \text{if } u \in \mathrm{fnv}(\pi) \\ \pi & \text{otherwise} \end{cases}$$

Fig. 3. Reduction semantics of CO_2 .

allows a participant A to perform an action in the session s containing γ (which, accordingly, evolves to γ'). Rule [Ask] allows A to proceed only if the contract γ at session s satisfies the property ϕ. The last three rules are mostly standard. In rule [Del] the label π fired in the premise becomes τ in the consequence, when π contains the delimited name/variable. This transformation is defined by the function $\mathrm{del}_u(\pi)$, where the set $\mathrm{fnv}(\pi)$ contains the free names/variables in π. For instance, $(x)\,\mathsf{A}[\mathsf{tell} \downarrow_x c.P] \xrightarrow{\mathsf{A}:\ \tau} (x)\,(\mathsf{A}[P] \ \mid\ \{\downarrow_x c\}_{\mathsf{A}})$. Here, it would make little sense to have the label $\mathsf{A}:\mathsf{tell} \downarrow_x c$, as x (being delimited) may be α-converted.

Implementing in Maude the semantics of CO_2 is almost straightforward [19]; here we show only the main rules (see [2] for the others). Rule `[Do]` uses the transition relation `=>` on bilateral contracts. Rule `[Ask]` exploits the Maude model checker to verify if the bilateral contract `g` satisfies the LTL formula `phi`. Rule `[Fuse]` uses the operator `|X|` to check compliance between the contracts `c` and `d`, then creates the session `s[A says c | B says d]` (with `s` fresh), and finally applies the substitution `{s / x}{s / y}` (delimitations are dealt with as in Fig. 3).

```
crl [Do] : A[do s a . P + P' | Q] | s[g] => {A : do s a} (A[P | Q] | s[g'])
           if g => {A says a} g' .

crl [Ask] : A[ask s phi . P + P' | Q] | s[g] => {A : ask s phi} A[P | Q]
            if g |- phi .

crl [Fuse] : (uVec , vVec) ({x c}A | {y d}B | S) => {K : fuse}
             (s , vVec) (s[A says c | B says d] | S{s / x}{s / y})
             if uVec == (x , y) / c |X| d / s := fresh(0 , S) .
```

4 Honesty

A remarkable feature of CO_2 is that it allows for writing *dishonest* agents which do not keep their promises. Intuitively, a participant is honest if she always fulfils her contractual obligations, in all possible contexts. Below we formalise the notion of honesty, by slightly adapting the one appeared in [3]. Then, we show how we verify in Maude a weaker notion, i.e. honesty *in a given context*.

We start by defining the set $\mathrm{O}^{\mathsf{A}}_s(S)$ of *obligations* of A at s in S. Whenever A is culpable at some session s, she has to fire one of the actions in $\mathrm{O}^{\mathsf{A}}_s(S)$.

Definition 5. *We define the set of atoms* $\mathrm{O}^{\mathsf{A}}_s(S)$ *as:*

$$\mathrm{O}^{\mathsf{A}}_s(S) \;=\; \left\{ \mathsf{a} \mid \exists \gamma, S' \,.\, S \equiv s[\gamma] \mid S' \text{ and } \gamma \xrightarrow{\mathsf{A}\ says\ \mathsf{a}} \!\!\!\!\rightarrow \right\}$$

We say that A *is* culpable at s in S *iff* $\mathrm{O}^{\mathsf{A}}_s(S) \neq \emptyset$.

The set of atoms $\mathrm{RD}^{\mathsf{A}}_s(S)$ ("Ready Do") defined below comprises all the actions that A can perform at s in one computation step within S (note that, by rule [Del], if s is a bound name then $\mathrm{RD}^{\mathsf{A}}_s(S) = \emptyset$). The set $\mathrm{WRD}^{A}_s(S)$ ("Weak Ready Do") contains all the actions that A may possibly perform at s after a finite sequence of transitions of A not involving any do at s.

Definition 6. *For all* S, A *and* s, *we define the sets of atoms:*

$$\mathrm{RD}^{\mathsf{A}}_s(S) = \left\{ \mathsf{a} \mid \exists S' \,.\, S \xrightarrow{\mathsf{A}:\ \mathsf{do}_s\,\mathsf{a}} S' \right\}$$

$$\mathrm{WRD}^{\mathsf{A}}_s(S) = \left\{ \mathsf{a} \mid \exists S' \,.\, S \xrightarrow{\mathsf{A}:\ \neq \mathsf{do}_s}{}^{*} S' \;\wedge\; \mathsf{a} \in \mathrm{RD}^{\mathsf{A}}_s(S') \right\}$$

where we write $S \xrightarrow{\mathsf{A}:\ \neq \mathsf{do}_s} S'$ *if* $\exists \pi.\ S \xrightarrow{\mathsf{A}:\ \pi} S' \;\wedge\; \forall \mathsf{a}.\ \pi \neq \mathsf{do}_s\,\mathsf{a}$.

A participant is *ready* if she can fulfil some of her obligations. To check if A is ready in S, we consider all the sessions s in S involving A. For each of them, we check that some obligations of A at s are exposed after some steps of A *not* preceded by other do_s of A. $\mathsf{A}[P]$ is honest *in a given system* S when A is ready in all evolutions of $\mathsf{A}[P] \mid S$. Then, $\mathsf{A}[P]$ is honest when she is honest in *all* S.

Definition 7 (Honesty). *We say that:*

1. S is A-free *iff it has no latent/stipulated contracts of* A, *nor processes of* A
2. A *is* ready in S *iff* $S \equiv (\boldsymbol{u})S' \wedge \mathrm{O}^{\mathsf{A}}_s(S') \neq \emptyset \implies \mathrm{WRD}^{\mathsf{A}}_s(S') \cap \mathrm{O}^{\mathsf{A}}_s(S') \neq \emptyset$
3. P *is* honest in S *iff* $\forall \mathsf{A} : (S \text{ is } \mathsf{A}\text{-free} \wedge \mathsf{A}[P] \mid S \to^{*} S') \implies \mathsf{A} \text{ is ready in } S'$
4. P *is* honest *iff, for all* S, P *is honest in* S

We have implemented items 2 and 3 of the above definition in Maude (item 4 is dealt with in the next section). CO_2 can simulate Turing machines [5], hence reachability in CO_2 is undecidable, and consequently WRD, readiness and honesty are undecidable as well. To recover decidability, we then restrict to finite state processes: roughly, these are the processes with neither delimitations nor parallel compositions under process definitions.

In Maude we verify readiness in a session s by searching if A can reach (with her moves only), a state which allows for a $\mathsf{do}_s\,\mathsf{a}$ move, for some a.

```
op ready? : Participant SessionName System Module -> Bool .
eq ready?(A,s,S,M:Module) =  metaSearch(M:Module, upTerm(< S > A s),
   '<_>__['S1:System , upTerm(A) , upTerm(s)],
   'S1:System => ''_'_['l:SLabel,'S2:System] /\
   '_:_[upTerm(A),'do__[upTerm(s),'a:Atom]] := 'l:SLabel,
   '*, unbounded, 0 ) =/= failure .
```

We start the search from the term `< S > A s`, whose meta-representation is obtained through the `upTerm` function. The search is performed according to the `A`-solo semantics of CO_2 (see Definition 10), which blocks all `do` at `s`. This is done by the operator `<_>__`. Then, we look for reachable systems `S1` where `A` can fire a `do` at `s`. If the search succeeds, `ready?` returns true. Note that if `A` has no obligations at `s` in `S`, `ready?` returns false — uncoherently with Def. 7. To correctly check readiness, we define the function `ready` (see [2]), which invokes `ready?` only when $\mathrm{O}^{\mathtt{A}}_{\mathtt{s}}(\mathtt{S}) \neq \emptyset$.

Verifying honesty in a context is done similarly. We use `metaSearch` to check that `A` is ready in all reachable states. The operator `<_>` gives the CO_2 semantics.

```
op search-honest-ctx : Participant System Module -> ResultTriple? .
eq search-honest-ctx(A,S,M:Module) = metaSearch(M:Module, upTerm(< S >),
  '<_>['S:System], 'ready[upTerm(A), 'S:System,'S:System, upTerm(M:Module)]
  = 'false.Bool, '*, unbounded, 0) .
op honest-ctx : Participant System Module -> Result .
ceq honest-ctx (A , S , M:Module) = true
  if search-honest-ctx (A , S , M:Module) == failure .
ceq honest-ctx (A , S , M:Module) = downTerm (T:Term , < (0).System > )
  if {T:Term,Ty:Type,S:Substitution} := search-honest-ctx (A,S,M:Module) .
```

Example 4. A travel agency `A` queries in parallel an airline ticket broker `F` and a hotel reservation service `H` in order to organise a trip for some user `U`. The agency first requires `U` to pay, and then chooses either to commit the reservation or to issue a refund (contract `CU`). When querying the ticket broker (contract `CF`), the agency first receives a quotation, and then chooses either to commit and pay the ticket, or to abort the transaction. The contract `CH` between `A` and `H` is similar.

```
eq CU = pay . (commit ; 0 (+) refund ; 0) .
eq CF = ticket . ( commitF ; payF ; 0 (+) abortF ; 0) .
eq CH = hotel . ( commitH ; payH ; 0 (+) abortH ; 0) .
```

In addition to the contracts above, the agency should respect the following constraints: (a) the agency refunds `U` only if both the transactions with `F` and `H` are aborted; (b) `A` pays the ticket and the hotel reservation only after it has committed the transaction with `U`; (c) either both the transactions with `F` or `H` are committed, or they are both aborted. A possible specification in Maude respecting the above constraints is given by the following process `P`:

```
eq P = ( xu , xf , xh ) ( tell xu CU . do xu pay .
       ( (tell xf CF . PF) | (tell xh CH . PH) | PU ) ) .

eq PF = do xf ticket . (do xh commitH . 0 + do xf abortF . 0) .
eq PH = do xh hotel  . (do xf commitF . 0 + do xh abortH . 0) .

eq PU = ask xh ([] ~ payH) . do xu refund . 0 +
        t . do xu commit . (do xf payF . 0 | do xh payH . 0) .
```

The process P first opens a session with U, and then advertises the contracts CF and CH, and in parallel executes PU. The process PF gets the ticket quotation, then either commits the hotel reservation, or aborts the flight reservation. Dually, PH gets the hotel quotation, then either commits the flight reservation, or aborts the hotel reservation. Note that the two choices in PF and PH ensure that constraint (c) above is satisfied: e.g., if PF fires the commitH (resp. abortF) prefix, the abortH (resp. commitF) branch in PH is disabled, and only commitF (resp. abortH) can be selected. The process PU checks if a refund is due to U. When the atom payH is no longer reachable in session xh, the ask passes, and the refund is issued. This guarantees constraint (a). In the τ-branch, PU commits the transaction with U, and then proceeds to pay both F and H. This satisfies constraint (b). Note that it may happen that PU chooses to commit even when CF or CH are not stipulated. Although this behaviour is conceptually wrong, it does not affect honesty. Indeed, honesty does not consider the domain-specific constraints among actions (e.g. (a), (b), (c) above), but only that the advertised contracts are respected.

We have experimented the function honest-ctx by inserting P in some contexts S where all the other participants U, F and H are honest (see [2] for details). The Maude model checker has correctly determined that P is honest in S.

```
red honest-ctx(A , S , ['TRAVEL-AGENCY-CTX]) .
rewrites: 53950741 in 38062ms cpu (38058ms real) (1417429 rewrites/second)
result Bool: true
```

Even though we conjecture that P is honest (in all contexts), we anticipate here that the verification technique proposed in § 5 does not classify P as honest. This is because the analysis is (correct but) not complete in the presence of ask: indeed, the precise behaviour of an ask is lost by the analysis, because it abstracts from the contracts of the context.

5 Model checking honesty

We now address the problem of automatically verifying honesty. As mentioned in § 1, this is a desirable goal, because it alerts system designers before they deploy services which could violate contracts at run-time (so possibly incurring in sanctions). Since honesty is undecidable in general [5], our goal is a verification technique which safely over-approximates honesty, i.e. it never classifies a process as honest when it is not. The first issue is that Def. 7 requires readiness to be preserved in all possible contexts, and there is an *infinite* number of such contexts. To overcome this problem, we present below an *abstract* semantics of CO_2 which preserves the honesty property, while neglecting the actual context where the process $\mathsf{A}[P]$ is executed.

The definition of the abstract semantics of CO_2 is obtained in two steps. First, we provide the projections from concrete contracts/systems to the abstract ones. Then, we define the semantics of abstract contracts and systems, and we relate the abstract semantics with the concrete one. The abstraction is always parameterised in the participant A the honesty of which is under consideration.

The abstraction $\alpha_{\mathsf{A}}(\gamma)$ of a bilateral contract $\gamma = \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d$ (Definition 8 below) is either c, or $\mathit{ctx}.c$ when d has a *ready*.

Definition 8. *For all γ, we define the abstract contract $\alpha_{\mathsf{A}}(\gamma)$ as:*

$$\alpha_{\mathsf{A}}(\mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d) \;=\; \begin{cases} c & \mathit{if}\ d\ \mathit{is\ ready\text{-}free} \\ \mathit{ctx}\ \mathsf{a}.c & \mathit{if}\ d = \mathit{ready}\ \mathsf{a}.d' \end{cases}$$

We now define the abstraction α_{A} of concrete systems, which just discards all the components not involving A, and projects the contracts involving A.

Definition 9. *For all A, S we define the* abstract system $\alpha_{\mathsf{A}}(S)$ *as:*

$$\begin{array}{lll} \alpha_{\mathsf{A}}(\mathsf{A}[P]) = \mathsf{A}[P] & \alpha_{\mathsf{A}}(s[\gamma]) = s[\alpha_{\mathsf{A}}(\gamma)] & \alpha_{\mathsf{A}}(\{\downarrow_x c\}_{\mathsf{A}}) = \{\downarrow_x c\}_{\mathsf{A}} \\ \alpha_{\mathsf{A}}(S \mid S') = \alpha_{\mathsf{A}}(S) \mid \alpha_{\mathsf{A}}(S') & \alpha_{\mathsf{A}}((u)S) = (u)(\alpha_{\mathsf{A}}(S)) & \alpha_{\mathsf{A}}(S) = \mathbf{0},\ \mathit{otherwise} \end{array}$$

Abstract semantics. For all participants A, the abstract LTSs $\xrightarrow{\ell}\!\!\!\!\rightarrow_{\mathsf{A}}$ and $\xrightarrow{\mu}_{\mathsf{A}}$ on abstract contracts and systems, respectively, are defined by the rules in Fig. 4. Labels ℓ are atoms, with or without the special prefix *ctx* — which indicates a contractual action performed by the context. Labels μ are either *ctx* or they have the form $\mathsf{A}\colon \pi$, where A is the participant in $\rightarrow_{\mathsf{A}}$, and π is a CO_2 prefix.

Rules for abstract contracts (first row in Fig. 4) are simple: in an internal sum, A chooses a branch; in an external sum, the choice is made by the context; in a *ready* $\mathsf{a}.c$ the atom a is fired. The rightmost rule handles a *ready* in the context contract. For abstract systems, some rules are similar to the concrete ones, hence we discuss only the most relevant ones. Rule [α-Do] involves the abstract transitions of contracts. The behaviour of abstract systems also considers context actions, labelled with *ctx*. If $c \vdash \phi$, then the $\mathsf{ask}\ \phi$ passes, indepedently from the context (rule [α-Ask]). If $c \nvdash \neg\phi$, then the $\mathsf{ask}\ \phi$ may pass or not, depending and the context (rule [α-AskCtx]). Rule [α-Fuse] says that a latent contract of A may always be fused (the context may choose whether this is the case or not). The context may also decide whether to perform actions within sessions ([α-DoCtx]). Unobservable context actions are modelled by rules [α-Ctx] and [α-DelCtx].

To check if $\mathsf{A}[P]$ is honest, we must only consider those A-free contexts not already containing advertised/stipulated contracts of A. Such systems will always evolve to a system which can be split in two parts: an A-*solo* system S_{A} containing the process of A, the contracts advertised by A and all the sessions containing contracts of A, and an A-*free* system S_{ctx}.

Definition 10. *We say that a system S is* A-solo *iff one of the following holds:*

$$S \equiv \mathbf{0} \qquad S \equiv \mathsf{A}[P] \qquad S \equiv s[\mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d] \qquad S \equiv \{\downarrow_x\ c\}_{\mathsf{A}}$$

$$S \equiv S' \mid S''\ \ \mathit{where}\ S'\ \mathit{and}\ S''\ \mathsf{A}\text{-}\mathit{solo} \qquad S \equiv (u)S'\ \ \mathit{where}\ S'\ \mathsf{A}\text{-}\mathit{solo}$$

We say that S is A-safe *iff $S \equiv (\boldsymbol{s})(S_{\mathsf{A}} \mid S_{ctx})$, with S_{A} A-solo and S_{ctx} A-free.*

The following theorems establish the relations between the concrete and the abstract semantics of CO_2. Theorem 3 states that the abstraction is *correct*, i.e. for each concrete computation there exists a corresponding abstract computation. Theorem 4 states that the abstraction is also *complete*, provided that a process has neither ask nor non-proper contracts.

$\mathsf{a}\,;\, c \oplus c' \xrightarrow{\mathsf{a}}\!\!\!\!\rightarrow_{\mathsf{A}} ctx\ \bar{\mathsf{a}}.c \quad \mathsf{a}\,.\, c + c' \xrightarrow{ctx:\bar{\mathsf{a}}}\!\!\!\!\rightarrow_{\mathsf{A}} ready\ \mathsf{a}.\ c \quad ready\ \mathsf{a}.\ c \xrightarrow{\mathsf{a}}\!\!\!\!\rightarrow_{\mathsf{A}} c \quad ctx\ \mathsf{a}.c \xrightarrow{ctx:\mathsf{a}}\!\!\!\!\rightarrow_{\mathsf{A}} c$

$$\frac{c \xrightarrow{\mathsf{a}}\!\!\!\!\rightarrow_{\mathsf{A}} c'}{\mathsf{A}[\mathsf{do}_s\ \mathsf{a}.P + P' \mid Q] \ \mid\ s[c] \ \xrightarrow{\mathsf{A:\ do}_s\ \mathsf{a}}_{\mathsf{A}} \ \mathsf{A}[P \mid Q] \ \mid\ s[c']} \quad [\alpha\text{-Do}]$$

$$\frac{s\ fresh}{(x)(\tilde{S} \ \mid\ \{\downarrow_x c\}_{\mathsf{A}}) \ \xrightarrow{ctx}_{\mathsf{A}} \ (s)(s[c] \ \mid\ \tilde{S}\{^s/_x\})} \quad [\alpha\text{-Fuse}]$$

$$\frac{c \vdash \phi}{\mathsf{A}[\mathsf{ask}_s\ \phi.P + P' \mid Q] \ \mid\ s[c] \ \xrightarrow{\mathsf{A:\ ask}_s\ \phi}_{\mathsf{A}} \ \mathsf{A}[P \mid Q] \ \mid\ s[c]} \quad [\alpha\text{-Ask}]$$

$$\frac{c \nvdash \neg\phi}{\mathsf{A}[\mathsf{ask}_s\ \phi.P + P' \mid Q] \ \mid\ s[c] \ \xrightarrow{ctx}_{\mathsf{A}} \ \mathsf{A}[P \mid Q] \ \mid\ s[c]} \quad [\alpha\text{-AskCtx}]$$

$$\frac{c \xrightarrow{ctx}\!\!\!\!\rightarrow_{\mathsf{A}} c'}{s[c] \xrightarrow{ctx}_{\mathsf{A}} s[c']}\ [\alpha\text{-DoCtx}] \qquad S \xrightarrow{ctx}_{\mathsf{A}} S\ [\alpha\text{-Ctx}] \qquad \frac{\tilde{S} \xrightarrow{ctx}_{\mathsf{A}} \tilde{S}'}{(u)\tilde{S} \xrightarrow{ctx}_{\mathsf{A}} (u)\tilde{S}'}\ [\alpha\text{-DelCtx}]$$

Fig. 4. Abstract LTSs for contracts and systems (full set of rules in [2]).

Theorem 3. *For all* A*-safe systems* S*, and for all concrete traces* η*:*

$$S \xrightarrow{\eta}{}^{*} S' \implies \exists \tilde{\eta} \ :\ \alpha_{\mathsf{A}}(S) \xrightarrow{\tilde{\eta}}{}_{\mathsf{A}}{}^{*} \alpha_{\mathsf{A}}(S')$$

Furthermore, if η *is* A*-solo and* S *is* ask*-free, then* $\eta = \tilde{\eta}$*.*

Theorem 4. *For all* ask*-free abstract system* $\tilde{S}$ *with proper contracts only:*

$$\tilde{S} \rightarrow_{\mathsf{A}}^{*} \tilde{S}' \implies \exists S, S'\ \mathsf{A}\text{-}safe.\ \alpha_{\mathsf{A}}(S) = \tilde{S} \ \wedge\ S \rightarrow^{*} S' \ \wedge\ \alpha_{\mathsf{A}}(S') = \tilde{S}'$$

The abstract counterparts of Ready Do, Weak Ready Do, and readiness are defined as expected, by using the abstract semantics instead of the concrete one (see [2] for details). The notion of honesty for abstract systems, namely α*-honesty*, follows the lines of that of honesty in Def. 7.

Definition 11 (α-honesty). *We say that* P *is* α-honest *iff for all* $\tilde{S}$ *such that* $\mathsf{A}[P] \rightarrow_{\mathsf{A}}^{*} \tilde{S}$*,* A *is ready in* $\tilde{S}$*.*

The main result of this paper follows. It states that α-honesty is a sound approximation of honesty, and — under certain conditions — it is also complete.

Theorem 5. *If* P *is* α*-honest, then* P *is honest. Conversely, if* P *is honest,* ask*-free, and has proper contracts only, then* P *is* α*-honest.*

In Maude, we implement abstract semantics for system and contracts for one-step transitions. We obtain their transitive closure, discarding labels, with the operator `<_>`. The function `ready` in `search-honest` computes abstract readiness.

```
op search-honest : Process Module -> ResultTriple? .
eq search-honest(P , M:Module) = metaSearch(M:Module, upTerm(< A[P] >),
   '<_>['S:System], 'ready['S:System,'S:System, upTerm(M:Module)]
   = 'false.Bool, '*, unbounded, 0) .

op honest : Process Module -> Result .
ceq honest (P, M:Module) = true if search-honest (P,M:Module) == failure .
ceq honest (P, M:Module) = downTerm (T:Term , < (0).System > )
    if {T:Term, Ty:Type, S:Substitution} := search-honest (P , M:Module) .
```

Honesty is checked by searching for states such that A is *not* ready. If the search fails, then A is honest. As in § 4, this function is decidable for finite state processes, i.e. those without delimitation/parallel under process definitions. The following example shows a process which was erroneously classified as honest in [5]. The Maude model checker has determined the dishonesty of that process, and by exploiting the Maude tracing facilities we managed to fix it.

Example 5. A store A offers buyers two options: clickPay or clickVoucher. If a buyer B chooses clickPay, A requires a payment (pay) otherwise A checks the validity of the voucher with V, an online voucher distribution system. If V validates the voucher (ok), B can use it (voucher), otherwise (no) B must pay. We specify in Maude the contracts CB (between A and B) and CV (between A and V) as:

```
eq CB  = clickPay . pay . 0 +
         clickVoucher . (- reject ; pay . 0 (+) - accept ; voucher . 0) .
eq CV = ok . 0 + no . 0 .
```

We can specify in Maude a CO_2 process for A as follows:

```
eq P = (x)(tell x CB . (do x clickPay . do x pay . 0 +
                        do x clickVoucher . ((y) tell y CV . Q))) .
eq Q = do y ok . do x - accept . do x voucher . 0 +
       do y no . do x - reject . do x pay . 0     + R .
eq R = t . (do x - reject . do x pay . 0) .
```

Variables x and y in P correspond to two separate sessions, where A respectively interacts with B and V. The advertisement of CV causally depends on the stipulation of the contract CB, because A must fire clickVoucher before tell y CV. In process Q the store waits for the answer of V: if V validates the voucher (first branch), then A accepts it from B; otherwise (second branch), A requires B to pay. The third branch R allows A to fire a τ action, and then reject the voucher. The intuition is that τ models a timeout, to deal with the fact that CV might not be stipulated. When we check the honesty of P with Maude, we obtain:

```
red honest(P , ['STORE-VOUCHER]) .
rewrites: 31649 in 72ms cpu (77ms real) (439545 rewrites/second)
result TSystem: < ($ 0,$ 1)(A[do $ 0 - reject . do $ 0 pay . (0).Sum] |
  $ 0[- accept ; voucher . 0(+)- reject ; pay . 0] | $ 1[ready ok . 0]) >
```

This means that the process P is dishonest: actually, the output provides a state where A is not ready. There, A must do ok in session y ($1), while A is only ready to do a -reject at session x ($0). This problem occurs when the branch R is chosen. To recover honesty, it suffices to replace R with the following process R':

```
eq R' = t . (do x - reject . do x pay . 0 | (do y no . 0 + do y ok . 0)) .
red honest(P' , ['STORE-VOUCHER]) .
rewrites: 44009 in 32ms cpu (30ms real) (1375195 rewrites/second)
result Bool: true
```

6 Conclusions

We have described an executable specification in Maude of a calculus for contract-oriented systems. This has been done in two steps. First, we have specified a model for contracts, and we have formalised in Maude their semantics, and the crucial notions of compliance and culpability (§ 2). This specification has been exploited in § 3 to implement in Maude the calculus CO_2 [4]. Then, we have considered the problem of honesty [5], i.e. that of deciding when a participant always respects the contracts she advertises, in all possible contexts (§ 4). Writing honest processes is not a trivial task, especially when multiple sessions are needed for realising a contract (see e.g. Ex. 4 and Ex. 5). We have then devised a sound verification technique for deciding when a participant is honest, and we have provided an implementation of this technique in Maude (§ 5).

Related work. Rewriting logic [12] has been successfully used for more than two decades as a semantic framework wherein many different programming models and logics are naturally formalised, executed and analysed. Just by restricting to models for concurrency, there exist Maude specifications and tools for CCS [17], the π-calculus [16], Petri nets [15], Erlang [14], Klaim [18], adaptive systems [7], *etc.* A more comprehensive list of calculi, programming languages, tools and applications implemented in Maude is collected in [13].

The contract model presented in § 2 is a refined version of the one in [5], which in turn is an alternative formalisation of the one in [8]. Our version is simpler and closer to the notion of *session behaviour* [1], and enjoys several desirable properties. Theorem 1 establishes that only one participant may be culpable in a bilateral contract, whereas in [5] both participants may be culpable, e.g. in A *says* $\mathsf{a}\,;\,c \mid \mathsf{B}$ *says* $\bar{\mathsf{a}}\,;\,d$. In our model, if both participants have an internal (or external) choice, then their contracts are *not* compliant, whereas e.g. $\mathsf{a}.c$ and $\bar{\mathsf{a}}.d$ (both external choices) are compliant in [5,8] whenever c and d are compliant. The exculpation property established by Theorem 2 is stronger than the corresponding one in [5]. There, a participant A is guaranteed to exculpate herself by performing (at most) two consecutive actions *of* A, while in our model two any actions (of *whatever* participant) suffice.

As far as we know, the concept of *contract-oriented computing* (in the meaning used in this paper) has been introduced in [6]. CO_2, a contract-agnostic calculus for contract-oriented computing, has been instantiated with several contract models — both bilateral [5,3] and multiparty [11,4]. Here we have instantiated it with the contracts in § 2. A minor difference w.r.t. [5,3,11] is that here we no longer have fuse as a language primitive, but rather the creation of fresh sessions is performed non-deterministically by the context (rule [Fuse]). This is equivalent to assume a contract broker which collects all contracts, and may establish sessions when compliant contracts are found. In [5], a participant A is considered honest when, in each possible context, she can always exculpate herself by a sequence of A-solo moves. Here we require that A is ready (i.e. some of her obligations are in the Weak Ready Do set) in all possible contexts, as in [3]. We conjecture that these two notions are equivalent. In [3] a type system has been

proposed to safely over-approximate honesty. The type of a process P is a function which maps each variable to a *channel type*. These are behavioural types (in the form of Basic Parallel Processes) which essentially preserve the structure of P, by abstracting the actual prefixes as "non-blocking" and "possibly blocking". The type system relies upon checking honesty for channel types, but no actual algorithm is given for such verification, hence type inference remains an open issue. In contrast, here we have directly implemented in Maude a verification algorithm for honesty, by model checking the abstract semantics in § 5.

References

1. F. Barbanera and U. de'Liguoro. Two notions of sub-behaviour for session-based client/server systems. In *PPDP*, 2010.
2. M. Bartoletti, M. Murgia, A. Scalas, and R. Zunino. Modelling and verifying contract-oriented systems in Maude. `http://tcs.unica.it/software/co2-maude`.
3. M. Bartoletti, A. Scalas, E. Tuosto, and R. Zunino. Honesty by typing. In *FMOODS/FORTE*, volume 7892 of *LNCS*, 2013.
4. M. Bartoletti, E. Tuosto, and R. Zunino. Contract-oriented computing in CO_2. *Sci. Ann. Comp. Sci.*, 22(1), 2012.
5. M. Bartoletti, E. Tuosto, and R. Zunino. On the realizability of contracts in dishonest systems. In *COORDINATION*, volume 7274 of *LNCS*, 2012.
6. M. Bartoletti and R. Zunino. A calculus of contracting processes. In *LICS*, 2010.
7. R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, and A. Vandin. Modelling and analyzing adaptive self-assembly strategies with Maude. In *WRLA*, volume 7571 of *LNCS*, 2012.
8. G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. *ACM Transactions on Programming Languages and Systems*, 31(5), 2009.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *TCS*, 2001.
10. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, 1998.
11. J. Lange and A. Scalas. Choreography synthesis as contract agreement. In *ICE*, 2013.
12. J. Meseguer. Rewriting as a unified model of concurrency. In *CONCUR*, volume 458 of *LNCS*, 1990.
13. J. Meseguer. Twenty years of rewriting logic. *JLAP*, 81(7-8), 2012.
14. M. Neuhäußer and T. Noll. Abstraction and model checking of core Erlang programs in Maude. *ENTCS*, 176(4), 2007.
15. M.-O. Stehr, J. Meseguer, and P. C. Ölveczky. Rewriting logic as a unifying framework for Petri nets. In *Unifying Petri Nets*, 2001.
16. P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0. *ENTCS*, 71, 2002.
17. A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. *ENTCS*, 71, 2002.
18. M. Wirsing, J. Eckhardt, T. Mühlbauer, and J. Meseguer. Design and analysis of cloud-based architectures with KLAIM and Maude. In *WRLA*, volume 7571 of *LNCS*, 2012.
19. T. F. Şerbănuţă, G. Roşu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207(2):305 – 340, 2009.

Value Iteration for Relational MDPs in Rewriting Logic*

Lenz Belzner

LMU Munich, PST Chair
belzner@pst.ifi.lmu.de

Abstract. Relational approaches to represent and solve MDPs exploit structure that is inherent to modelled domains in order to avoid or at least reduce the impact of the *curse of dimensionality* that propositional representations suffer from. By explicitly reasoning about relational structure, policies that lead to specific goals can be derived on a very general, abstract level; thus, these policies are valid for numerous domain instantiations, regardless of the particular objects participating in it. This paper describes the encoding of relational MDPs in terms of rewrite theories by encoding non-deterministic domain dynamics as rewrite rules. Narrowing is employed to solve these relational MDPs symbolically. Resulting abstract value functions are simplified through state subsumption, which is realized by matching abstract state terms.

1 Introduction

The framework of *Markov decision processes* (MDPs) allows to model domains with non-deterministic action outcomes and arbitrary reward functions, thus serving well for modelling problems of *sequential decision making under uncertainty* [1]. Various exact and approximate techniques to solve MDPs exist, such as *value iteration*, *policy iteration* and *modified policy iteration* [2]. Given a reward specification (i.e. system goals), a solution of a MDP can be computed that is either a value function mapping states to their corresponding expected values (according to the reward function of the MDP) or a policy mapping states to actions that are maximizing the expected reward as the policy is executed.

Algorithms for solving MDPs suffer from the *curse of dimensionality* [3], rendering them infeasible for large-scale domains. To overcome this problem, effort has been made to exploit inherent structure of domains by employing factored or relational, first-order representations of states and actions instead of propositional ones, allowing to represent structured problem domains more concisely. Thus, computation becomes feasible also for larger domains, but the additional complexity that arises from structured domain representations has to be taken into account when solving the according MDP [4,5,6].

Rewriting logic is a formal logical framework that lends itself naturally to modelling non-deterministic and concurrent domains on a symbolic level [7,8].

* This work has been partially funded by the EU project ASCENS, 257414.

It provides support for formally specifying structured domains through modularization and object-orientation, as well as explicit sorting and sub-sorting. As a reflective logical framework, it also provides formal definitions to operations on the meta-level, as for example introduction or renaming of variables, which are viable operations for symbolic programming in general, and for symbolic dynamic programming in particular. In rewriting logic, logical deduction and program execution are considered equal, a fact that lead to the implementation of the language MAUDE that allows to execute formal specifications given in rewriting logic straightforwardly [9].

Rewriting logic offers a straightforward approach to specify the nondeterministic nature of MDPs through the use of rewrite rules and their corresponding semantics. Also, the concepts of matching and narrowing offer powerful and efficient approaches to relate symbolic terms to each other, thus allowing to model and execute abstract state subsumption and regression on the first-order level. This paper describes the specification of relational MDPs in rewriting logic, and how matching [10] and narrowing [11,12] can be employed to solve them using first-order abstraction and avoiding propositionalization. In particular, this paper introduces a model-based dynamic programming algorithm for relational MDPs that employs first-order reasoning and computes exact solutions with the following properties:

- It operates on explicitly sorted state representations, allowing domain specifications to define sort hierarchies and operation polymorphism.
- The Bellman backup is performed regressive; only goal-relevant abstract states are constructed and evaluated.
- Given a particular reward function, the resulting value function (and the corresponding policy) are optimal w.r.t. reward function and domain model regardless of any particular state an agent finds itself in.
- Constants are only introduced where they are goal-relevant. Thus, reasoning and regression are performed on the first-order level wherever possible.
- The combined state-action space is factored, leading to concise results.
- Partial goal specifications are supported.

The formalization of the algorithm as a rewrite theory directly provides a specification of a corresponding MAUDE program; thus, an implementation of the algorithm is provided as well.

This paper is outlined as follows: Section 2 discusses in more detail value iteration to solve MDPs exactly as well as the rewriting logic framework and the concepts of matching and narrowing. Section 3 describes how relational MDPs can be encoded in terms of a rewrite theory and how rewriting logic concepts can be used to solve these relational MDPs symbolically. Section 4 discusses an example to illustrate the approach described in this paper. Finally, section 5 compares the approach to related work, summarizes the results described in this paper and hints at possibilities for further research.

2 Preliminaries

This section introduces the MDP framework (section 2.1) and value iteration as well as rewriting logic and the concepts of rewriting and narrowing (section 2.2).

2.1 MDPs and Value Iteration

Definition 1. *A* Markov decision process (MDP) *is a tuple* (S, A, T, γ, R) *with* S *a set of states,* A *a set of actions,* $T : S \times A \times S \to \mathbb{R}$ *a transition function,* $\gamma \in [0; 1]$ *a discount factor and* $R : S \to \mathbb{R}$ *a reward function.*[1]

Definition 2. *A* relational MDP *employs a representation for* S *and* A *that allows for existential quantification of first-order variables and establishment of relations between domain objects* (see e.g. [13]).

A tuple as given in definiton 1 specifies the non-deterministic, discrete time dynamics of a domain in terms of a transition system. The transition function T encodes the probability that executing an action $a \in A$ in a particular state $s \in S$ will result in a state $s' \in S$; note that s and s' may be equal, indicating absence of an action effect. The discount factor γ reflects how much an agent prefers immediate over long-term rewards; the smaller γ is chosen, the more immediate rewards will impact behaviour of an agent acting according to the MDP. The reward function defines incentives that are given to the agent in particular states; in other words, it specifies which states are valuable to achieve.

Definition 3. *A* value function $V : S \to \mathbb{R}$ *maps states to values. The value of a state* $s \in S$ *is the reward gained in* s *plus the expected discounted future reward when acting greedily w.r.t.* V.

Definition 4. *A* policy $\pi : S \to A$ *maps states to actions. An agent* acting according to a policy π *executes action* $\pi(s)$ *when being in state* s.

Solving a MDP means to compute either a value function V mapping states to expected values w.r.t. the given reward function R, or to provide a policy π that maps states in S to actions in A that are going to maximize the expected reward of an agent acting according to π. In both cases, the solution of the MDP can be used by an agent to determine which action to execute in which state in order to gather as much reward as possible in the long run.

$$V(s) = R(s) + \gamma \max_{a \in A} \left(\sum_{s' \in S} T(s, a, s') V(s') \right) \tag{1}$$

Equation (1) shows the *Bellman equation* [3], that defines the actual value for each state in an MDP according to the transition function T and the reward

[1] Rewards may also be specified action-wise, $R : S \times A \times S \to \mathbb{R}$. The approach to value iteration discussed in section 3 could be extended to also deal with this representation.

function R. The general idea is that the value of a state is the sum of the reward this state will expose to the agent and the expected discounted future reward when moving on to the next state by executing an action that is assumed to be optimal w.r.t. the value function V.

$$V_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A} \left(\sum_{s' \in S} T(s, a, s') V_i(s') \right) \tag{2}$$

For an MDP that has n states, this would result in n equations, one for each state, with n unknowns. As the max-operation is non-linear, solving these equations is problematic. To this end, Bellman proposed an iterative approach, called *value iteration*. Equation (2) shows the iteration step known as *Bellman backup*. Value iteration is guaranteed to converge to the optimal solution [3]. The value function V can be arbitrarily initialized, a common approach is to set $V_0(s) = R(s)$. Iteration is performed until $|V_{i+1}(s) - V_i(s)| < \epsilon(1-\gamma)/\gamma$ for each state $s \in S$ and a given error bound $\epsilon \in \mathbb{R}$. This ensures that the maximum difference of V_{i+1} to the real value function V is smaller than ϵ for all states [2].

2.2 Rewriting Logic

Rewriting logic is suited towards the formal specification of non-deterministic, concurrent systems. System states are encoded in user-definable terms that are constructed from specified operations that can be enhanced with axioms like associativity, commutativity or idempotency. System dynamics are then described in terms of so-called rewrite rules, that allow to specify non-deterministic and concurrent behaviour.

The core element to describe systems in rewriting logic are *rewrite theories*, i.e. tuples of the form $(\Sigma, E \cup A, R)$, where Σ contains sorts and operations that are used to construct state terms, E is a set of equations that define equivalence classes for these terms, A is a set of axioms like associativity, commutativity or idempotency specified for operations in Σ, and R is a set of rewrite rules that define the system dynamics. A rewrite theory represents a transition system, where states are terms in Σ, and rewrite rules in R define state transitions.

A central concept in rewriting logic is *matching*, which is performed *modulo axioms* and *with extensions* (denoted by $:=_{\mathrm{Ax}}$). Consider an infix operation $\circ$ being associative and commutative, i.e. rendering terms constructed by means of $\circ$ into a *multiset*. Then, for example, the term $a \circ b$ matches modulo axioms with extensions the term $a \circ c \circ b$, as the latter has the same equivalence class as the term $a \circ b \circ c$ (due to commutativity) and $a \circ b$ is a subterm of $a \circ b \circ c$.

While state terms in Σ specify the static representation of a system, its dynamics are formalized in terms of *rewrite rules*. Those are of the form

$$label: \ t \ \rightarrow \ t' \ \textbf{if} \ Conditions$$

where t and t' are terms of the same kind and may contain free sorted variables. If t matches modulo axioms with extensions a given subject term, the

subject term's matched portions are rewritten to t'. A rule's label may be omitted. Rewrite rules can optionally be conditional, in which case rewriting only is applied if all conditions evaluate to true. Conditions can occur in four different forms:

1. Sort or kind tests, denoted $s : Sort$.
2. Equational conditions constrain variable values through an equation $u = u'$. Equality is computed according to $E \cup A$ of the rewrite theory.
3. Matching conditions $u := u'$ that constrain the syntactic structure of a variable in t (modulo axioms). Matching conditions can be used to define locally scoped variables (if u is a free variable not occurring in t).
4. Rewrite conditions $u \to u'$, that evaluate to true if u' can be derived by rewriting u according to rewrite rules in R. Free variables not occurring in t may be introduced in u'.

As a rule may match different portions of a subject term, this representation of system dynamics offers a natural way to model concurrency by rewriting a term on various positions simultaneously according to one or multiple rewrite rules. On the other hand, non-determinism is expressed if (partially or completely) overlapping portions of a subject term match one or more rewrite rules, i.e. if different applications of rewrite rules are possible without being applicable concurrently. In this case, a subject term evolves non-deterministically in any possible way. For example, consider the rewrite rules $(i) : a \to a'$, $(ii) : a \to a''$ and $(iii) : b \to b'$. Then, the term $a \circ b$ can be rewritten to $a' \circ b'$ by applying rules (i) and (iii), and also to the form $a'' \circ b'$ (using rules (ii) and (iii)).

While rewriting treats variables in a rewriting problem universally quantified, i.e. answering a problem of the form $\forall \boldsymbol{x} : t(\boldsymbol{x}) \to_? t'(\boldsymbol{x})$, a technique called *narrowing* deals with corresponding problems where variables are treated existentially, i.e. $\exists \boldsymbol{x} : t(\boldsymbol{x}) \to_? t'(\boldsymbol{x})$, representing *symbolic reachability* problems. To answer queries of this form, instead of matching rules and subject terms as in rewriting, they are *unified* in order to perform narrowing, meaning that variables in both terms may be instantiated to achieve syntactic term unification. I.e., when narrowing, rewrite rules are applied if (one or more subterms of) the subject term can be unified with a rule's lefthand side. Note that, when narrowing, righthand sides of rewrite rules may contain variables not specified in their lefthand side, allowing rewrite rules to introduce fresh variables. For an in-depth discussion of rewriting logic and the concepts of matching, rewriting and narrowing see for example [9].

3 Value Iteration for Relational MDPs in Rewriting Logic

This section discusses how to encode relational MDPs in rewriting logic and how to solve these MDPs symbolically using the concepts of matching and narrowing.

3.1 Relational MDPs in Rewriting Logic

In order to perform symbolic value iteration, a relational representation of the underlying MDP has to be specified. To this end, a relational MDP (S, A, T, γ, R) is encoded as a rewrite theory $(\Sigma, E \cup A, R)$. To avoid representation of states in a propositional manner, relations between domain objects can be specified, providing a symbolic representation of states and actions. This allows for a concise representation of MDPs even when there is a big number of domain objects. Also, as will be shown in section 3.2, a relational MDP can be solved completely symbolically, only grounding variables where this is relevant for goal reachability.

Domain objects are simply represented by corresponding sorts. Properties and relations of theses objects that change due to domain dynamics are encoded by parametrizable *fluents* representing truth values of properties of a particular state, which are encoded by a corresponding sort FLUENT. Subsorts of FLUENT can be specified as necessary to further structure the state space, allowing to restrict fluent variables to particular domain objects, for which sorts can also be specified in Σ. Negation of fluents (i.e. the explicit absence of a particular state property) is represented by an operation $\neg : \text{FLUENT} \to \text{FLUENT}$. The state space is constructed in terms of a sort STATE with $\text{FLUENT} < \text{STATE}$ by an associative and commutative operation $\wedge : \text{STATE} \times \text{STATE} \to \text{STATE}$ that is representing logical conjunction. A constant *false* is defined for sort STATE to denote constraint violations, and $\neg F \wedge F = \mathit{false}$ for all $F \in \text{FLUENT}$.

State terms that are syntactically constructed in this way may still contain semantic inconsistencies. Semantic constraints (e.g. state invariants) can be specified in terms of confluent and terminating equations that reduce states that violate constraints to *false*. States that violate constraints (e.g. state invariants) will be completely reduced to *false*: $S \wedge \mathit{false} = \mathit{false}$ for all $S \in \text{STATE}$.

The set of equivalence classes of state terms can then be considered the set S of states of a relational MDP. Note that each equivalence class may render different instances of state terms equal according to their relational structure, thus providing a representation of abstract first-order states.

Primitive actions executable by agents are encoded by a parametrizable sort ACTION. Equivalence classes on ACTION-terms then form the set of actions A of a relational MDP. As for state terms, free variables are allowed in action terms. The action space is thus raised to an abstract level that allows to exploit its relational structure, especially when taking into account relations between state and action space, e.g. if state and action terms share free variables.

Example 1. Consider a signature with sorts TRUCK, BOX and CITY and the sorts FLUENT, STATE and ACTION as above. One can then for example define a fluent $\mathit{boxOn} : \text{BOX} \times \text{TRUCK} \to \text{FLUENT}$. Consider e.g. fluents *truckIn* and *boxIn* defined similarly. Then $\mathit{truckIn}(t, c) \wedge \mathit{boxOn}(b, t) \wedge \mathit{boxIn}(b', c)$ is a STATE-term in Σ. [2]

[2] The following notational conventions are introduced, unless stated otherwise: Lowercase letters represent terms (that may contain free variables), uppercase letters represent free variables. In particular, $t, t', T, T' \in \text{TRUCK}$, $b, b', B \in \text{BOX}$ and $c, c', C, C' \in \text{CITY}$ represent constants and free variables denoting domain objects;

An action representing a truck loading a box can be defined by an operation $load : \text{TRUCK} \times \text{BOX} \to \text{ACTION}$. Then, $load(t, b)$ is an ACTION-term in Σ.

The fact (i.e. state constraint) that a truck can only be in one city at a time can be specified by a conditional equation $truckIn(T, C) \wedge truckIn(T, C') \wedge S = false$ **if** $C \neq C'$.

To encode the relation of states, actions (e.g. an optimal action in a particular state) and any corresponding values (e.g. a state's probability to be reached or its expected value), a sort SAV-TUPLE is defined to represent *state-action-value* tuples (SAV-Tuples) of the form (s, a, v). When either a or v are omitted, the relation is valid for all actions or values, respectively. Note that SAV-tuples may relate state and action terms that share variables, consider for example a SAV-tuple $(s(\boldsymbol{x} \cup \boldsymbol{y}), a(\boldsymbol{y} \cup \boldsymbol{z}), v)$ where state and action share the variables $\boldsymbol{y}$. Thus, SAV-tuples allow to exploit structure of the combined state-action space.

To model the transition function T of a MDP, a rewrite rule can be defined for any transition $T(s, a, s') = p$ to specify this transition in a rewrite theory:

$$(s, a) \to (s', p) \ .$$

In order to match or unify a subject SAV-tuple with the rule's lefthand side in MAUDE, s and s' have to include a free variable of sort STATE (e.g. $s \wedge S$). This encoding also explicitly shows the solution to the frame problem [14].

If executing an action a in a state s exposes non-deterministic outcomes, the according transitions of T can be encoded in terms of disjunctive rewrite rules (with $\sum p_i = 1$; and considering $\vee$ as disjunctive constructor for sets of SAV-tuples):

$$(s, a) \to (s'_1, p_1) \vee (s'_2, p_2) \vee ... \vee (s'_n, p_n) \ .$$

The state terms s and the s'_i can be considered as pre- and postconditions of action a. This representation of domain dynamics provides a solution to the *frame problem* [14], avoiding the necessity to specify all parts of the state that remain unchanged by action execution.

Example 2. Consider action *load* from example 1. If a truck executing this action succeeds to load a box (supposing the truck is in the same city as the box) with a probability of 0.9, and fails to load it with a probability of 0.1, these transitions are expressed in terms of the following rewrite rule:

$$\begin{aligned}(&truckIn(T, C) \wedge boxIn(B, C) \wedge S,\ load(T, B)) \to \\ &(truckIn(T, C) \wedge boxOn(B, T) \wedge S,\ 0.9) \\ &\quad \vee (truckIn(T, C) \wedge boxIn(B, C) \wedge S,\ 0.1) \ .\end{aligned}$$

An MDP's reward function R is represented in terms of an operation mapping states to values and appropriate equations, e.g. $reward(boxIn(b, c) \wedge S) = 1.0$. Reward is considered to be zero for all other states.

$s, s', s'_i \in$ STATE and $a, a' \in$ ACTION represent state and action terms, $S \in$ STATE denotes a free state variable; $p, p_i, v, v' \in \mathbb{R}^3$ denote probabilities of transitions and values of states, respectively.

3.2 Symbolic Value Iteration

This section discusses how to perform symbolic value iteration in order to solve relational MDPs specified as described in section 3.1, thus providing a relational representation of a value function and policy that is optimal w.r.t. a given reward function. This requires definition of a regression operation, summation of values for non-deterministic action effects, maximization of regressed states to optimal actions, and value discounting as well as reward distribution.

Regression through Narrowing. According to the definition of value iteration as in equation (2), the value of executing an action a in a state s is computed according to the values of the states that are reachable from s when performing a. When recalling the specification of a relational MDP from section 3.1, the set of states is the set of equivalence classes over state terms. Many of those terms may describe states from which a particular goal state (that will provide a reward to an agent) is not reachable at all according to the rewrite rules from the MDP specification, thus it is necessary to identify state-terms from which a goal is reachable. Also, actions have to be instantiated with appropriate variables related to these states in order to factor the action space properly. In order to only compute the value for the states and actions that are relevant for goal reachability, these state-action terms are computed backwards from a given goal (i.e. a given value function). This backward construction is called *regression*.

To allow for regressive induction of state-action space abstractions from given goal states, rewrite rules that specify domain dynamics are transformed into regressive rewrite rules. The key idea is to define for a given state from which preceding states it can be reached by execution of a particular action, and with what probability this action will lead to the given state. The value of the reached state is then used to compute values of preceding states according to transition probabilities.

Consider a relational MDP (S, A, T, γ, R), a value function $V : S \to \mathbb{R}$, and a rewrite theory $(\Sigma, E \cup A, R)$ encoding the MDP as outlined in section 3.1. Then, the regressive specification for the dynamics of an action can be given by inverting the (possibly disjunctive) rewrite rule in R that specifies an effect for this action (which is of the form $(s, a) \to (s'_1, p_1) \vee ... \vee (s'_n, p_n)$) for each of the effects specified in its righthand side, i.e. for each $i \in [1, ..., n]$:

$$(s'_i, V(s'_i)) \to (s, a, V(s'_i) * p_i * \gamma) \ .$$

Example 3. Consider the rewrite rule for action *load* from example 2. Considering $V \in$ FLOAT already encoding the value for particular abstract states, the regressive rewrite rules for action *load* are:

$$\begin{aligned} &(\mathit{truckIn}(T, C) \wedge \mathit{boxOn}(B, T) \wedge S,\ V) \to \\ &(\mathit{truckIn}(T, C) \wedge \mathit{boxIn}(B, C) \wedge S,\ \mathit{load}(T, B),\ V * 0.9 * \gamma) \ . \\ &(\mathit{truckIn}(T, C) \wedge \mathit{boxIn}(B, C) \wedge S,\ V) \to \\ &(\mathit{truckIn}(T, C) \wedge \mathit{boxIn}(B, C) \wedge S,\ \mathit{load}(T, B),\ V * 0.1 * \gamma) \ . \end{aligned}$$

For value iteration, this representation of domain dynamics serves two purposes. First, it allows to compute from a given value function V all abstract (i.e. relational) states from which a particular state $s' \in Domain(V)$ is reachable by narrowing (for a single step) a SAV-tuple (s', v) with $v = V(s')$ according to the inverted rewrite rules of a rewrite theory encoding a MDP. Note that this narrowing step produces a set of SAV-tuples $(s, a, v * p * \gamma)$ that also encode the action a that, when executed in state s, would result in state s' with probability p. Second, with regard to value iteration as in equation (2), this narrowing step resembles computation of $\gamma * T(s, a, s')V_i(s')$ when computing $V_{i+1}(s)$.

As narrowing is employed, variable grounding is (only) applied where necessary (i.e. where relevant to the reachable state s') through unification of the state that is currently regressed with the lefthand sides of rewrite rules specifying domain dynamics. Also, if the state to be regressed misses any action postconditions, these are induced to regressed state terms by unification if the subject term to be regressed and action effect rules' lefthand sides contain a free state variable (i.e. are of the form $s \wedge S$). Thus, also partially specified goal states can be regressed. If system goals are specified in V_0 (e.g. by instantiating V_0 with a set of SAV-tuples $(s, R(s))$, thus resembling the MDP's reward function R), narrowing exactly grounds variables and induces fluents that are relevant for an optimal policy w.r.t. these goals. Note that regression may lead to states that contain inconsistencies. States that violate constraints (see section 3.1) will subsequently be ignored by further computation.

Summation of Non-Deterministic Action Effects. Regressing the SAV-tuples of a given value function computes a set of SAV-tuples (s, a, v) denoting the states s from which the states in the value function domain can be reached through execution of action a. While v already incorporates transition probabilities and known state values, it does not yet take into account that an action may have multiple outcomes.

In equation (2), this fact is addressed by the summation of expected values of all states that are reachable by execution of a particular action a, weighted by transition probabilities. Given the set of SAV-tuples as computed by regression, this summation can be performed in a rewrite theory in terms of an equation:

$$(s, a, v) \vee (s, a, v') = (s, a, v + v') \ .$$

Maximization through Abstract State Subsumption. After regressing the set of SAV-tuples encoding all states and actions that lead to abstract states in the domain of the current value function (taking into account non-deterministic action effects), it has to be ensured that only optimal actions for each abstract state remain in the new value function V_{i+1}. This is achieved by only keeping in the set of SAV-tuples those elements that exhibit the maximal value that is gained through action execution for each possible state. As states are relational, they may overlap or even subsume other states completely. To deal with subsumption, the concept of matching can directly be employed to model state subsumption, as a more general term matches a more concrete one.

Definition 5. *A state* s subsumes *a state* s' *iff the state term of* s *matches modulo axioms with extension the state term of* s'*.*

For maximization over the set of actions, a state s' with value v' is removed from the regressed set of SAV-tuples if it is subsumed by another state s with greater or equal value $v \geq v'$:

$$(s, a, v) \vee (s', a', v') = (s, a, v) \textbf{ if } s :=_{\mathrm{Ax}} s' \wedge v \geq v' \ .$$

Example 4. The state $boxOn(B, T)$ subsumes $boxOn(b, t)$, as the former, more general term matches the latter. Note that, as matching is performed *with extension* when testing for subsumption, for example the state $boxOn(b, t)$ subsumes $boxOn(b, t) \wedge boxOn(b', t')$.

Now consider $(boxOn(b, t) \wedge S, a, 1.0)$ and $(boxOn(B, T) \wedge S, a', 2.0)$ being SAV-tuples in the set to be maximized. In this case, it is clearly preferable to execute action a' when *any* box is on *any* truck, because the expected value of this action is 2.0. Thus, the former tuple can be dropped for further computation.

Finally, consider $(boxOn(b, t) \wedge S, a, 3.0)$ and $(boxOn(B, T) \wedge S, a', 2.0)$ being maximized. Then the former should not be dropped, as action a is preferable if exactly box b is on truck t; otherwise, if another box or another truck are involved, action a' should be executed. Both tuples are necessary to deduce this behaviour.

Value Iteration & Decision List Policies. To complete a value iteration step according to equation (2) after performing maximization, the currently gained reward for all states in the set of SAV-tuples has to be distributed according to the MDP's reward function R.

New SAV-sets resembling a relational MDP's value function are iteratively constructed by abstract state regression, summation of state values taking into account non-deterministic action effects, maximization of expected value for states and actions, discounting and reward distribution. For each iteration i, the resulting SAV-set exactly resembles V_{i+1} in value iteration according to equation (2) for all $s \in S$ from which states in the domain of V_i are reachable. States that are not covered by the set are assigned a value of zero, thus the SAV-set can be considered a function. Iteration stops if for all $(s, a, v) \in V_{i+1}$ there exists a $(s, a, v') \in V_i$ such that $|v - v'| < \epsilon(1 - \gamma)/\gamma$ for a given error bound $\epsilon \in \mathbb{R}$.

The resulting SAV-set representing the converged optimal value function V (with an error bounded by ϵ) can then be interpreted as a decision list, sorted by values of the SAV-tuples it contains; thus, overlapping and subsuming states are dealt with. An agent can then traverse the list elements, checking whether its current situation matches with a state term from one of the SAV-tuples in the list. If so, it should execute the action of the particular SAV-tuple. This way, the agent will always execute the action that has the maximal expected reward in the current state. I.e., the decision list resembles a policy $\pi : S \to A$ for the MDP that was solved with the presented algorithm, considering $\pi(s) = a \Leftrightarrow (s, a, v) \in V \wedge \forall (s, a', v') \in V : v \geq v'$ and $\pi(s)$ being any action (e.g. *noop*) if $\nexists\, a, v : (s, a, v) \in V$.

4 Example

Consider the BOXWORLD domain [4] where trucks have to deliver boxes to particular cities. Trucks can load a box if they are in the same city, unload a box if they loaded it before, and they can drive from one city to another. All actions succeed with a probability of 0.9, otherwise, they have no effect. Sorts for domain objects are defined as in example 1. Note the explicit specification of an action *noop*, indicating absence of action execution in a particular state.

Constants:

$op\ box : \rightarrow \text{BOX}.\ op\ city : \rightarrow \text{CITY}\ .$

Relations:

$op\ boxOn : \text{BOX} \times \text{TRUCK} \rightarrow \text{FLUENT}\ .\ op\ boxIn : \text{BOX} \times \text{CITY} \rightarrow \text{FLUENT}\ .$
$op\ truckIn : \text{TRUCK} \times \text{CITY} \rightarrow \text{FLUENT}\ .$

Actions:

$op\ load : \text{TRUCK} \times \text{BOX} \rightarrow \text{ACTION}\ .\ op\ unload : \text{TRUCK} \times \text{BOX} \rightarrow \text{ACTION}\ .$
$op\ driveTo : \text{TRUCK} \times \text{CITY} \rightarrow \text{ACTION}\ .\ op\ noop : \rightarrow \text{ACTION}\ .$

Variables:

$vars\ T : \text{TRUCK},\ B : \text{BOX},\ C, C' : \text{CITY},\ S : \text{STATE}\ .$

Constraints:

$ceq\ boxOn(B,T) \wedge boxOn(B,T') = false\ \textbf{if}\ T \neq T'\ .$
$ceq\ boxIn(B,C) \wedge boxIn(B,C') = false\ \textbf{if}\ C \neq C'\ .$
$ceq\ truckIn(T,C) \wedge truckIn(T,C') = false\ \textbf{if}\ C \neq C'\ .$
$eq\ boxOn(B,T) \wedge boxIn(B,C) = false\ .$

Effects:

$$[noop] : (S, noop) \rightarrow (S, 1.0)\ .$$
$$[load] : (truckIn(T,C) \wedge boxIn(B,C) \wedge S, load(T,B)) \rightarrow (truckIn(T,C) \wedge boxOn(B,T) \wedge S, 0.9) \vee (truckIn(T,C) \wedge boxIn(B,C) \wedge S, 0.1)\ .$$
$$[unload] : (truckIn(T,C) \wedge boxOn(B,T) \wedge S, unload(T,B)) \rightarrow (truckIn(T,C) \wedge boxIn(B,C) \wedge S, 0.9) \vee (truckIn(T,C) \wedge boxOn(B,T) \wedge S, 0.1)\ .$$
$$[driveTo] : (truckIn(T,C) \wedge S, driveTo(T,C')) \rightarrow (truckIn(T,C') \wedge S, 0.9) \vee (truckIn(T,C) \wedge S, 0.1)\ .$$

From this specification, regression operators for action effects can be compiled as shown in example 3. Consider a reward function giving a reward of 1.0 to all states subsumed by $boxIn(box, city) \wedge S$, and zero reward to all other states. The initial SAV-set V_0 to be iteratively refined then consists of only one SAV-tuple $(boxIn(box, city) \wedge S, 1.0)$. Note that this goal does not explicitly include a complete postcondition of any of the actions specified for the domain. Fluents that are relevant for an optimal value function will be induced by narrowing when performing regression.

Considering $\gamma = 0.9$, regressing and discounting V_0 as discussed in section 3.2 yields the following SAV-set. Note the relations (i.e. fluents) and fresh variables induced to the SAV-tuples by narrowing when unifying the SAV-tuple to be

regressed with effect rules' lefthand sides. Also note the regression to states that introduce relations for objects that are not relevant for goal reachability, but correctly deduced from the domain specification. These states are subsequently subsumed by the goal state when performing maximization.

1. $(boxIn(box, city) \wedge S,\ noop,\ 0.9)$
2. $(boxIn(box, city) \wedge truckIn(T, city) \wedge S,\ load(T, box),\ 0.09)$
3. $(boxIn(box, city) \wedge truckIn(T, C) \wedge S,\ driveTo(T, C'),\ 0.09)$
4. $(boxIn(box, city) \wedge truckIn(T, C') \wedge S,\ driveTo(T, C),\ 0.81)$
5. $(boxOn(box, T) \wedge truckIn(T, city) \wedge S,\ unload(T, box),\ 0.81)$
6. $(boxIn(box, city) \wedge boxIn(B, C) \wedge truckIn(T, C) \wedge S,\ load(T, B),\ 0.09)$
7. $(boxIn(box, city) \wedge boxIn(B, C) \wedge truckIn(T, C) \wedge S,\ load(T, B),\ 0.81)$
8. $(boxIn(box, city) \wedge boxOn(B, T) \wedge truckIn(T, C) \wedge S,\ unload(T, B),\ 0.09)$
9. $(boxIn(box, city) \wedge boxOn(B, T) \wedge truckIn(T, C) \wedge S,\ unload(T, B),\ 0.81)$

Value summation due to non-deterministic effects is applied to SAV-tuples 3 and 4[4], 5 and 6 as well as 7 and 8. Maximization then removes all SAV-tuples but 1 and 5 (as tuple 1 subsumes all but tuple 5 and has a higher value). Finally, reward distribution leads to the following SAV-set for V_1.

1. $(boxIn(box, city) \wedge S,\ noop,\ 1.9)$
2. $(boxOn(box, T) \wedge truckIn(T, city) \wedge S,\ unload(T, box),\ 0.81)$

With $\epsilon = 0.05$, subsequent iterations yield the following result; it can be interpreted as a decision list sorted by value in order to be employed as an optimal policy by an agent. By matching a particular state the agent finds itself in with the states in the list from top to bottom, the optimal action (that yields the highest expected reward when executed in the situation at hand) can be determined.

1. $(boxIn(box, city) \wedge S,\ noop,\ 9.95)$
2. $(boxOn(box, T) \wedge truckIn(T, city) \wedge S,\ unload(T, box),\ 8.85)$
3. $(boxOn(box, T) \wedge truckIn(T, C) \wedge S,\ driveTo(T, city),\ 7.17)$
4. $(boxIn(box, C) \wedge truckIn(T, C) \wedge S,\ load(T, box),\ 6.38)$
5. $(boxIn(box, C) \wedge truckIn(T, C') \wedge S,\ driveTo(T, C),\ 5.16)$

5 Conclusion

This section discusses related approaches, summarizes symbolic value iteration in rewriting logic and hints at further research possibilities.

[4] Note that summation of values for non-deterministic effects as outlined in section 3.2 would not apply for SAV-tuples 3 and 4, as their states expose different variable names. Careful variable renaming or replacing syntactic equality by cross-subsumption of states when summing non-deterministic effects cure this issue.

5.1 Related Work

The first successful approach to solve MDPs with value iteration completely on the symbolic level was achieved by Symbolic Dynamic Programming (SDP) [4]. It uses the situation calculus [15] to represent first-order MDPs, thus allowing for full first-order logic quantifications for variables. While the situation calculus is a very expressive specification language, the frame problem has to be addressed explicitly in the specification of domain dynamics, in contrast to specifications in rewriting logic. Another difference of SDP to the approach presented in this paper is that dynamics are defined in a regressive manner and per fluent (in terms of so-called *successor state axioms*) and not per action, thus diverging from modern software design paradigms as for example object-orientation, where dynamics are typically defined in terms of operations. As a consequence, compilation of regressive successor state axioms from progressive, operation-oriented specifications becomes a complex transformation. Also, because of the complexity of regressed state formulas, consistency checking and simplification is a very complex task. Even if these tasks are manageable automatically in theory, the authors of SDP only reported on a preliminary implementation that illustrated their approach, but simplification of regression results was applied manually.

The fluent calculus [16] can be considered a progressive counterpart to the situation calculus as it represents states as associative-commutative terms of fluents. First-order value iteration for the fluent calculus (FOVIA) [17,5] can be performed in a fully automated manner due to restricted expressivity of the fluent calculus when compared to the situation calculus, as only existential quantification of variables is allowed. As in the presented approach, FOVIA uses a notion of state subsumption that allows for simplification of regressed symbolic value functions. FOVIA also uses AC1-unification to regress states to their predecessors, but this unification procedure is not parametrizable with an equational theory as when using narrowing, thus restricting expressivity for MDPs that can be solved with FOVIA. Also, the algorithm cannot natively deal with reward states that include a particular number of resources, like e.g. $boxIn(B, C) \wedge boxIn(B', C)$, as this formula reduces to $boxIn(B, C)$ when quantifying variables existentially in first order logic. As rewriting logic employs membership equational logic, reward states exposing this structure can be natively dealt with employing the presented approach to solve relational MDPs with rewriting logic. Finally, the specification of relational MDPs in terms of rewrite theories allows for incorporation of rewriting logic features as for example sort-hierarchies, polymorphism, object-oriented system representation or meta-level operations [9].

In previous work, rewriting logic has been employed to implement action programming in rewriting logic [18]. There are two main distinctions to symbolic value iteration: First, though also dealing with uncertainty, action programming does not necessarily depend on MDPs as domain model. Second, action programming in rewriting logic is *progressive*, expanding possible world dynamics from an initial state. On the contrary, value iteration is a regressive task.

5.2 Summary

This paper discussed the representation of relational MDPs in rewriting logic and how to use the concepts of matching and narrowing to solve them symbolically, resulting in advantages regarding computational effort and expressivity when compared to propositional solution techniques. To this end, states were represented as associative-commutative fluent terms containing variables for domain objects. A regression operator was introduced by exploiting the capabilities of narrowing, allowing for symbolic computation and instantiation of variables to particular objects where this is goal-relevant. As narrowing unifies goal-states and rule specifications, goal-relevant properties of the state space are also induced automatically, thus allowing for computation of optimal value-functions for only partially specified goals. Simplification of the resulting symbolic value function was performed through state subsumption, which was realized by symbolically matching associative-commutative state terms. By relating variables in state and action terms, both state and action space are partitioned properly.

As the specified rewrite theory directly implies a MAUDE program, an implementation was straightforward.[5] The approach was illustrated using the implementation on an example.

5.3 Further Work

As rewriting logic is also intended to model concurrent systems, it would be interesting to leverage the presented symbolic value iteration technique with rewriting logic representations and concepts of concurrency to be able to deal with multi-agent domains, incorporating specification of parallel, eventually collaborative actions and agent behaviour synchronization.

Recently, a generalization algorithm for rewrite theories has been introduced [19]. This technique could be used to learn the domain dynamics from perception samples gathered by an agent operating in an unknown environment. Generalization could be employed to deduce a MDP's transition function in terms of rewrite rules that resemble the observed dynamics of a system, thus refining at run-time a potentially restricted or incorrect design-time specification. Gathered knowledge could subsequently be used to optimize an agent's behaviour by solving a learned MDP at run-time.

References

1. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. 1st edn. John Wiley & Sons, Inc., New York, NY, USA (1994)
2. Russell, S.J., Norvig, P.: Artificial Intelligence - A Modern Approach (3. internat. ed.). Pearson Education (2010)
3. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton, NJ, USA (1957)

[5] The implementation is available at `http://www.pst.ifi.lmu.de/~belzner/odin/`.

4. Boutilier, C., Reiter, R., Price, B.: Symbolic dynamic programming for first-order MDPs. In Nebel, B., ed.: IJCAI, Morgan Kaufmann (2001) 690–700
5. Hölldobler, S., Skvortsova, O.: A logic-based approach to dynamic programming. In: Proceedings of the Workshop on Learning and Planning in Markov Processes–Advances and Challenges at the Nineteenth National Conference on Artificial Intelligence (AAAI04). (2004) 31–36
6. Raghavan, A., Joshi, S., Fern, A., Tadepalli, P., Khardon, R.: Planning in factored action spaces with symbolic dynamic programming. In Hoffmann, J., Selman, B., eds.: AAAI, AAAI Press (2012)
7. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1) (April 1992) 73–155
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theor. Comput. Sci. **285**(2) (2002) 187–243
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L., eds.: 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)
10. Eker, S.: Fast matching in combinations of regular equational theories. Electr. Notes Theor. Comput. Sci. **4** (1996) 90–109
11. Escobar, S., Meseguer, J., Thati, P.: Narrowing and rewriting logic: from foundations to applications. Electr. Notes Theor. Comput. Sci. **177** (2007) 5–33
12. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: Unification and narrowing in maude 2.4. In Treinen, R., ed.: RTA. Volume 5595 of Lecture Notes in Computer Science., Springer (2009) 380–390
13. Sanner, S., Boutilier, C.: Approximate linear programming for first-order MDPs. In: UAI, AUAI Press (2005) 509–517
14. Mccarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Machine Intelligence. Volume 4. (1969) 463–502
15. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. Illustrated edn. The MIT Press, Massachusetts, MA (2001)
16. Thielscher, M.: Introduction to the fluent calculus. Electron. Trans. Artif. Intell. **2** (1998) 179–192
17. Großmann, A., Hölldobler, S., Skvortsova, O.: Symbolic dynamic programming within the fluent calculus. In: Proceedings of the IASTED International conference on Artificial and Computational Intelligence. (2002) 378–383
18. Belzner, L.: Action programming in rewriting logic. TPLP **13**(4-5-Online-Supplement) (2013)
19. Alpuente, M., Escobar, S., Meseguer, J., Ojeda, P.: A modular equational generalization algorithm. In Hanus, M., ed.: Logic-Based Program Synthesis and Transformation. Volume 5438 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 24–39

Maude-PSL: Reconciling Intuitive and Formal Specification in Cryptographic Protocol Analysis

Andrew Cholewa[1], Fan Yang[1], Catherine Meadows[2], and José Meseguer[1]

[1] University of Illinois at Urbana-Champaign, USA
{acholew2, fanyang6, meseguer}@illinois.edu
[2] Naval Research Laboratory, Washington, DC, USA
catherine.meadows@nrl.navy.mil

Abstract. Maude-NPA is a narrowing-based model checker for analysing cryptographic protocols in the Dolev-Yao model modulo equations. Currently, Maude-NPA relies on a strand-based notation that, while very precise, is less familiar to users of the Alice-Bob notation, and is rather difficult to read and write. Therefore, we propose a new language, called the Maude Protocol Specification Language(Maude-PSL). The Maude-PSL extends the Alice-Bob notation with the following additional pieces of information: how each principal interprets the messages it sends and receives, the information each principal is assumed to know at the start of the protocol execution, and the information the principal should know after execution. The Maude-PSL also simplifies Maude-NPA syntax for specifying intruder capabilities and attack states. The semantics of the language is defined as a translation into the strand-based model using rewriting logic, providing a formal grounding in a well understood model of cryptographic protocols.

1 Extended Abstract

Throughout the history of the formal analysis of cryptographic protocols, there has been a tension between readability and precision. On the one hand, there is the message-passing paradigm, also referred to as the Alice-Bob notation. In the message-passing paradigm, we define a protocol globally as a numbered sequence of message communications. We write each step using the notation $A \to B : M$, which says that A sends the message M to B. This approach gives us a clear understanding of how the protocol is supposed to behave. However, this notation is meant to be read by experts in cryptographic protocol analysis, and relies on a variety of implicit assumptions, and the expertise of the readers to keep things simple. Unfortunately, this can lead to ambiguities. For example, consider the protocol step $A \to B : enc(K, M)$ where A is sending a message M encrypted with key K to B. This single step could mean either that B has the key K, and decrypts $enc(K, M)$, or that B does not have K, in which case B cannot interpret the message and simply passes it on to a third party.

In order to address this problem, various role-based approaches have been developed, which endeavor to be more precise than the Alice-Bob notation. In a

role-based approach, we define a protocol locally by specifying the actions taken by each principal at each step of the protocol. For example, each principal may have a sequence of messages that it sends and receives. We may also specify a set of tests that the principal performs on each received message. This type of approach specifies exactly what is happening at each step of the protocol. However, this approach also tends to obscure the intended behavior of the protocol. As a result, we run the risk of specifying a protocol that does not actually express what we have in mind.

In an effort to have the best of both worlds, we propose a new input language for Maude-NPA (a symbolic model checker for verifying cryptographic protocols, see [1]): the Maude Protocol Specification Language (Maude-PSL). The Maude-PSL extends the Alice-Bob notation with additional information typically specified in role-based specification methodologies. In particular, we split each message into two forms: the version of the message understood by the sender, and the version understood by the receiver. For example, consider the following toy protocol:

1. $A \to S : \mathrm{enc}(K, M)$
2. $S \to B : \mathrm{enc}(K, M)$

where $\mathrm{enc}(K, M)$ is the encryption of message M with key K, which is known only by A and B. In the Maude-PSL, we write

1. $A \to S : \mathrm{enc}(K, M) \vdash N$
2. $S \to B : N \vdash \mathrm{enc}(K, M')$

where $\mathrm{enc}(K, M)$ is the version of the message $\mathrm{enc}(K, M)$ as seen by A (since A built the message). Since S does not know K, S cannot decrypt the message. Therefore, all S knows is that he/she has received some message N from A, which he/she then forwards to B. Meanwhile, because B knows the key K, B can decrypt the message N. Therefore, B knows that the message he/she receives from S is the encryption of some message M' with the key K. This is an improvement over the current specification language for Maude-NPA, which requires the user to manually translate the protocol into a strand-based notation that is fairly distant from the Alice-Bob notation in which protocols are typically formulated.

The Maude-PSL allows us to specify the information that each role should know before and after a session of the protocol. This is not supplied by Maude-NPA, but can be useful in formulating possible attacks, composing the protocol with another protocol, and understanding the protocol's purpose.

The Maude-PSL possesses a flexible, Maude-based syntax for specifying the term structure, type structure, and algebraic properties of a protocol. The semantics of the language is defined as a translation into the strand-based model (see [2]) using rewriting logic, providing a formal grounding in a well understood model of cryptographic protocols.

The Maude-PSL can be found at http://maude.cs.uiuc.edu/tools/Maude-NPA/Maude-PSL

Acknowledgements

We thank Santiago Escobar for his helpful suggestions about these ideas. This work has been supported in part by NSF grant CNS 13-19109 and by an NRL contract.

References

1. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA, Version 2.0. (2011)
2. Fábrega, F.J.T., Herzog, J.C., Guttman, J.D.: Strand spaces: Proving security properties correct. Journal of Computer Security **7** (1999) 191–230

Formalization and Verification of BPMN Models using Maude

Nissreen El-Saber[1,2] and Artur Boronat[2]

[1] Faculty of Computers and Informatics, Zagazig University, Egypt
[2] Department of Computer Science, University of Leicester, UK
{nase1,aboronat}@le.ac.uk

Abstract. OMG's Business Process Model and Notation (BPMN) standard provides an informal specification of a technology-independent modelling language for designing business processes. However, BPMN models may include structural issues that hinder their design. In this paper, we propose a formal characterization and semantics specification of well-formed BPMN processes in rewriting logic using Maude with a focus on data-based decision gateways and data objects semantics. Our formal specification adheres to the BPMN standards and enables model checking using Maude's LTL model checker. An example of compliance checking with the CMMI Configuration Management process area is presented.

Keywords: BPMN, Maude, compliance, LTL model checking

1 Introduction

Business processes (BPs) can be modelled using different notations, using either formal or informal representations. From a Computer Science point of view, formal modelling languages are more reliable and verifiable, while from a business point of view, non-technical users usually prefer to use graphical modelling languages due to their accessibility. The Business Process Model and Notation (BPMN) is a widely-used notation for modelling BPs in the early stages of systems life cycle. According to the OMG [1], 72 implementations of the BPMN are reported for known businesses (for example, Oracle). There are a number of issues with the BPMN standards that allow for ambiguity and unstructured BP models [2, 3, 4]: (1) the unclear semantics of different BPMN elements makes it possible to end up with incompatible interpretations for designing, analyzing and using BP models [2], (2) the formalization and use of data objects is under-represented, although they are considered as resources (e.g. [3, 4, 5, 6]), (3) according to the standards, conditions affecting flow divergence are defined as part of the transition flow connecting other objects, while gateways can accommodate these conditions and their evaluation as part of their logical behaviour. These issues can be handled and/or avoided using our proposed approach by providing more strict requirements for well-formed BPMN models. In this work,

a relevant excerpt of BPMN elements that is used regularly[3] is formalized in Maude [8] and the notion of well-formed BPMN processes is introduced and applied in order to guarantee some convenient structural properties.

The proposed formalization provides a mechanism to formalize data objects and the effect of business rules on their state according to contextual conditions and/or dependency relationships. Moreover, we propose a novel mechanism to represent and evaluate guard conditions in decision gateways. Our formalization proposes a new implementation for inclusive OR merge gateways that allow synchronization and prevent deadlocks, which may occur in other approaches (e.g. [3, 9, 6]). The formalization introduced in this paper uses Maude [8], as the formalization language. In addition to the expressiveness of its underlying logic for concurrent systems, Maude allows for defining formal executable specifications for other languages, or formalisms. Moreover, Maude has a verification toolkit (e.g. Maude LTL Model Checker [10]) which can be used to formally analyse and verify the models in Maude with respect to different LTL properties [8]. This makes the tandem formed by rewriting logic [11] and its Maude implementation a very convenient setting for formalizing BPMN models. The paper is organized as follows: an overview of BPMN, Maude and the example used throughout the paper are presented in Section 2. Section 3 presents the proposed formalization of a subset of the BPMN syntax in Maude. This is followed by the notion of well-structured and well-formed BPMN processes. Then we propose the behaviour specifications for BPMN elements in Section 5 followed by proposing a compliance checking approach using Maude LTL model checker in Section 6. Section 7 presents the related work, conclusions and future work.

2 Preliminaries

This section provides a brief idea about the tools and notions we use afterwards; i.e. BPMN and Maude, as well as introducing the example used throughout the paper. BPMN is a standard modelling notation for representing BPs in the design phase of systems development. BPMN 2.0 has five main categories of elements: flow nodes, connecting flow elements, swimlanes, data and artefacts. We focus on an excerpt of the BPMN elements which is graphically represented in Fig. 1. The BPMN main elements are the flow nodes (i.e. activities, events, gateways), connecting flows (e.g. sequence flows, associations), the data objects, and artefacts (e.g. text annotations used in Fig. 1 to mark other elements).

Definition 1. *(**BPMN Model**) A BPMN Model O is a tuple (OS, FO, A, E, G, DO, T, SF, MF, ASSC, TS, SP, E_S, E_I, E_E, ANDgates, XORgates, ORgates), where:*

- $OS = FO \cup DO$*; i.e. OS is the set of flow objects FO and data objects DO,*
- $FO = A \cup E \cup G$*; i.e. FO is the set of activities A, events E, and gateways G,*

[3] The BPMN 2.0 [1] defines 50 constructs and their attributes. However, less than 20% of its vocabulary is used regularly in designing BP models [7].

- $A = TS \cup SP$; *i.e. A is the set of tasks* TS *and sub-processes* SP,
- $E = E_S \cup E_I \cup E_E$; *i.e. E is the set of start events* E_S, *intermediate events* E_I *and end events* E_E,
- $G = ANDgates \cup XORgates \cup ORgates$; *i.e. G is the set of AND gateways* $ANDgates$, *XOR gateways* $XORgates$, *and OR gateways* $ORgates$,
- $T = SF \cup MF \cup ASSC$; *i.e. T is the set of connecting objects "transitions" of sequence flow* SF, *message flow* MF *and associations* $ASSC$.

Maude is a high-performance term rewriting engine that provides support for both equational and rewriting logic specification and programming of concurrent systems in particular [8]. The specifications in Maude are executable theories in rewriting logic [11], which is a flexible logical framework for expressing a wide range of concurrency models and distributed systems [10]. A Maude's functional module is a theory $\mathcal{R}=(\Sigma, E)$ in membership equational logic (`MEL`), where the algebraic signature Σ is a set of declarations of sorts, subsorts and function symbols and E is the set of conditional equations $t = t'$ **if** *cond* and conditional membership axioms $t : s$ **if** *cond* stating that the term t has sort (i.e., data type) s when *cond* holds. A Maude's system module M specifies a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \phi, R)$ where $(\Sigma, E \cup A)$ is the membership equational theory specified by the signature equational attributes and equations and membership statements in the module, A is a set of axioms, so that both the equations E and the rules R are applied modulo the axioms A, ϕ is the function specifying the frozen arguments of each operator in Σ (c.f [8]), and R is a set of rewrite rules which may be conditional (i.e., `crl` [Label] : $t \Rightarrow t'$ **if** $Cond$). The rules describe all the local transitions (i.e., state changes) in the system. In addition, Maude represents the functions as operators, i.e. `op`f`:`$\mathtt{s}_1 \ldots \mathtt{s}_n$`->s` where f is the function name, $\mathtt{s}_1 \ldots \mathtt{s}_n$ are the arguments sorts and `s` is the result (function) sort. Maude uses functions `reduce` to reduce the terms to its canonical form using the equations and function `rewrite` to execute the specifications through the equations and rules to reach a canonical form of the term with its resulting sort. In this paper, we use Maude system modules which include the BPMN syntax specifications as operators and equations as well as the semantics specifications as rewrite rules. Specifically we are encoding BPMN elements as objects (o_i) using Maude general representation of objects [12] (`<OID:CID|AS>`), where `OID` is the object identifier, `CID` is the corresponding class identifier, and `AS` is the set of object's attributes.

In order to give a better explanation of the proposed formalization, we introduce an example in Fig. 1 which will be used throughout the paper. It represents a process model called *Release Baseline*, a subprocess of the Configuration Management (CM) system, where a configuration item (CI) is an entity designated for one or more related work products such as tangible assets (e.g. hardware) and intangible assets (e.g. software, OS) [13]. A collection of CIs that are used in a project or a company may be baselined whenever they are sufficiently stable, enabling a more strict control for changing them. In the process *Release baselines*, authorization is checked in order to modify the baselines using the document *Authorization List*. If the authorization is granted, a change request

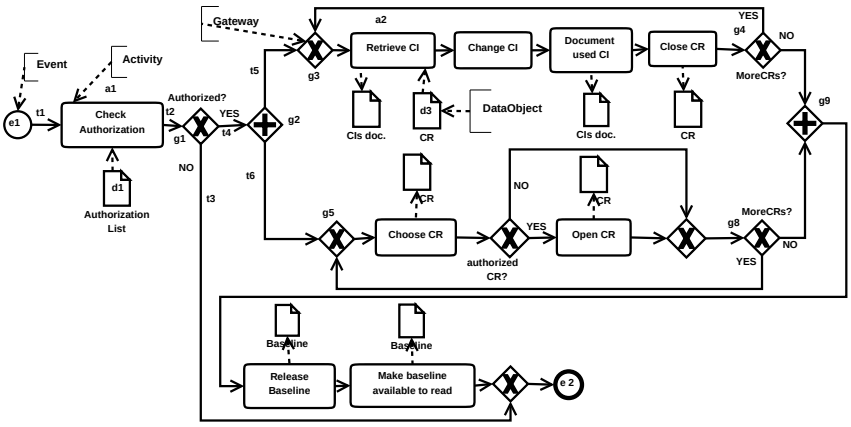


Fig. 1. Release Baseline Process - BPMN representation

(CR) is chosen and it is opened to implementation (`open CR`). While a specific CR is open, the related CI is retrieved, changed, and documented before the CR is closed. If there is more than one CR waiting, the same procedure is repeated until no more CRs are left in the process. After that the baseline is released and made available to the stakeholders. We are assuming the process has one (`Authorization List`), one configuration item (`CI doc`), three change requests (`CR-1`, `CR-2`, `CR-3`) and one (`Baseline`) data objects.

3 BPMN Abstract Syntax

In this section we propose a formal syntax for BPMN elements and an algebraic representation in Maude. Object identifiers (`OID`) for activities, events, gateways, data objects and connecting transitions are represented by the symbols: (ai, ei, gi, di, ti where $i \in \mathbb{N}$). The `CID` identifies the object's type by using pre-defined operators of the main sorts; such as (`task`) for task activity, (`aforkgate`) for an AND fork gateway. Each BPMN element has the sort `Object`, and a process, which is a set of these objects, is of sort `ObjectSet`. The relations is represented by means of sub-sorting relation (`subsort Object < ObjectSet`). Hence, a *process state* is the term representing the set of objects in an `ObjectSet`. In Fig. 1, event (`e1`), task (`a1`), data object (`d3`), and AND fork gateway (`g2`) are examples of BPMN objects. In the proposed formalization, these objects are represented as follows:

```
< e1 : starteEvent | eventType : start ; in : notrans ; out : t 1 >,
< a1 : task | name : "Check Authorization" ; in : t 1 ; out : t 2 >,
< d3 : dataObject | name : "CR" ; linkedObject : a 2 ; status : open >,
< g2 : aforkgate | in : t 4 ; out : (t 5,t 6) >
```

The attributes of each object represent its properties, e.g., attribute `name` for String name of the object, and attribute `status` for the pre-defined status of the data object (e.g., `open`). The relations between the objects in the process

are implicitly represented in the object itself using attributes `in` and `out` which contain the transition identifiers for the input flow (from the predecessor(s)) and output flow (to the successor(s)) respectively. The transitions connect the objects and represented by unique *transitions* (of sort `TransSymbol`) and `notrans` is the identity element. The data object `d3` should be linked to an activity to use it (i.e., attribute `linkedObject`).

```
op name‘:_ : String -> Attribute .
ops in‘:_ out‘:_ : TransSymbol -> Attribute .
op linkedObject‘:_ : ActivitySymbol -> Attribute .
```

BPMN gateways are modelled as constructor operators of type `GateCid` (e.g, `aforkgate`, `xsplitgate`, `omergegate`). The terms *split/fork* are used for divergence (e.g. `g2` in Fig. 1) and *join/merge* to refer to the combination of two or more parallel or alternative paths into one path respectively (e.g. `g9` in Fig. 1). For the decision gateways (XOR and OR), to control the divergence, a guard is defined as part of the gateway itself (i.e. attribute). In this paper we will present guards with one single expression for each output flow, however, the guard syntax can be extended to represent a larger class of expressions. The guard is linked to a transition which is marking the branch it should follow. Therefore, the guard is a set of pairs, where each pair contains the guard expression and the associated transition. The first part of the pair is an `Expression`, and the second part is the associated `TransSymbol` linking the succeeding flow. The `Expression` has the form (`Var opSymbol Val`), where the `Var` is the variable name defined by the modeller, the `Val` is the (Number or String) value assigned to the variable, and the `opSymbol` is one of logical comparison symbols (`<=`,`>=`,`<`,`>`) for less than or equal, greater than or equal, less and greater than with numeric values and (`==`,`=/=`) equal or not equal for numeric or String values. The operator `(_,_)` is used to define this in Maude. The associativity and commutativity properties of the guard expression is captured by the operator `_._` while `noexp` is the guard identity element.

```
op (_,_) : Expression TransSymbol -> Gexp .
op noexp : -> Gexp .
op _._ : Gexp Gexp -> Gexp [ctor assoc comm id: noexp] .
```

The guard of gateway `g1` in Fig. 1 is represented as `(Authorized?=="YES", t4).(Authorized?=="NO",t3)`. There is a value (*control value*) which is compared with the guard values (i.e. `YES`, `NO`). This value is entered by the modeller and modelled as an attribute `controlValues` of sort `CV`. The operator that puts together the control values is (`op controlValues‘:_ : CV -> Attribute [ctor assoc comm id:noCV]`). The syntax proposed above still allows for defining *spaghetti* design of BPs, making their formalization tedious and error prone. Hence, in the next section, a list of structural and well-formedness properties is introduced.

4 Well-Formed BPMN Processes

We introduce a set of useful functions which are used later in the formal definitions. Function $|\_|$, defined as $(|\_| : set \rightarrow \mathbb{N})$, takes a set and returns the number of its elements. Functions Oid and Cid are defined as $(Oid : Object \rightarrow OID)$ and $(Cid : Object \rightarrow CID)$ to return the object identifier and the class identifier for an input object. Functions in and out are defined as $(in : Object \rightarrow T)$ and $(out : Object \rightarrow T)$, where T is the set of connecting flows defined in Definition 1. They return the input and output transition flows for an object. Function $eventType$ is defined as $(eventType : Object \rightarrow TypeofEvent)$, where it returns the value of the `eventType` attribute in an event object. Function $itsSplit$ returns the relevant split gateway object's identifier for a given merge gateway object, and is defined as $(itsSplit : Object \rightarrow OID)$. Finally function $preds$ returns the set of object predecessors starting from an input object until the start event (or a boundary exception event) is reached (defined as $preds : Object\ ObjectSet \rightarrow ObjectSet$) and $succs$ returns the set of object successors starting from the input object until the end event is reached (defined as $succs : Object\ ObjectSet \rightarrow ObjectSet$). The following requirements, extracted from [1], are introduced in order to obtain a structured BPMN processes in Definition 2. In particular, (1) start and exception events have no incoming flows and have one outgoing flow, (2) an end event has no outgoing flows and has one incoming flow, (3) a process that has an end event, should have a start event.

Definition 2. *(**Well-Structured Process**) A well-structured business process* $\texttt{S-BPMN} = OS$ *iff : (1)* $\forall o_i \in OS\ (Cid(o_i) \in \{\texttt{startEvent}, \texttt{exception}\} \rightarrow (|in(o_i)| = 0 \wedge |out(o_i)| = 1))$*, (2)* $\forall o_i \in OS\ (Cid(o_i) = \texttt{endEvent} \rightarrow |in(o_i)| = 1 \wedge |out(o_i)| = 0)$*, and (3)* $\forall o_i \in OS\ (Cid(o_i) = \texttt{endEvent} \rightarrow \exists o_j \in OS\ (Cid(o_j) = \texttt{startEvent}))$*.*

Based on that, a BP model can have more than one start/end event, a split gateway without a corresponding merge gateway, a gateway to have multiple incoming and outgoing transitions at the same time, which considered ambiguity representation, and causes many structural and semantic problems in the models. Our formalization provides a more precise definition for the accepted models; i.e. the well-formed BPMN models (adapted from [6]). A well-formed BP is a well-structured BP which satisfies the following properties: (1) a process have one start event and one end event, (2) activities and non-exception intermediate events have one input and one output transitions, (3) fork and decision gateways have one input transition and at least two output transitions, (4) join and merge gateways have one output transition and at least two transitions as inputs, (5) except for `exception` events, each split gateway has a corresponding merge gateway from the same type, forming a *block* in the model. Exception objects attached to an activity boundary split the flow and then the flow is merged with the normal flow using an XOR merge gateway, and (6) every object is in a *connected path* from the start or an exception event to the end event. This is formalized in the following definition.

Definition 3. ***(Well-formed BPMN model)****: A well-structured BPMN process (`S-BPMN`) is a Well-Formed Model (`W-BPMN`) iff*

1. $\exists o_i \in OS\ (Cid(o_i) = \texttt{startEvent} \rightarrow \nexists o_j \in OS\ (Cid(o_j) = \texttt{startEvent}) \wedge i \neq j$, *and* $\exists o_i \in OS\ (Cid(o_i) = \texttt{endEvent} \rightarrow \nexists o_j \in OS\ (Cid(o_j) = \texttt{endEvent}) \wedge i \neq j$, *(function* `wfstartend`*),*
2. $\forall o_i \in A \cup E_I \backslash \texttt{exception}\ (\exists t_n, t_m \in T\ (in(o_i) = t_n \wedge in(o_i) = t_m \wedge t_n \neq t_m \neq \texttt{notrans})$*, (functions* `wfException`*,*`wfActivities`*),*
3. $\forall o_i \in OS\ (Cid(o_i) \in \{\texttt{ANDfork,XORsplit,ORsplit}\} \rightarrow |in(o_i)| = 1 \wedge |out(o_i)| > 1)$*, (function* `wfGates`*),*
4. $\forall o_i \in OS\ (Cid(o_i) \in \{\texttt{ANDjoin,XORmerge,ORmerge}\} \rightarrow |in(o_i)| > 1 \wedge |out(o_i)| = 1)$*, (function* `wfGates`*),*
5. *The gateways block structure: (function* `wfGates`*),*
   (a) $\forall o_i \in OS\ (Cid(o_i) = \texttt{aforkgate} \rightarrow \exists o_j \in OS\ (Cid(o_j) = \texttt{ajoingate} \wedge itsSplit(o_j) = Oid(o_i)))\wedge$
   (b) $\forall o_i \in OS\ (Cid(o_i) = \texttt{xsplitgate} \rightarrow \exists o_j \in OS\ (Cid(o_j) = \texttt{xmergegate} \wedge itsSplit(o_j) = Oid(o_i)))\wedge$
   (c) $\forall o_i \in OS\ (Cid(o_i) = \texttt{osplitgate} \rightarrow \exists o_j \in OS\ (Cid(o_j) = \texttt{omergegate} \wedge itsSplit(o_j) = Oid(o_i)))\wedge$
   (d) $\forall o_i \in OS\ (eventType(o_i) = \texttt{exception} \rightarrow \exists o_j \in OS\ (Cid(o_j) = \texttt{xmergegate} \wedge itsSplit(o_j) = Oid(o_i)))$*, and*
6. $\forall o_i \in OS(\exists o_j, o_k \in OS\ (((Cid(o_j) = \texttt{startEvent} \vee eventType(o_j) = \texttt{exception}) \wedge Cid(o_k) = \texttt{endEvent}) \rightarrow (o_j \in preds(o_i, OS) \wedge o_k \in succs(o_i, OS)))$*, (function* `wfpath`*).*

In point (5), gateways are required to be designed as a *block* in the model, i.e. each split gateway should have an accompanying merge gateway of the same type. A block has only one entrance point and one exit point, e.g. the split gateway input flow and the merge gateway output flow respectively in the case of acyclic models (check gateways g2 and g9 in Fig. 1). Notice that the definitions above does not exclude the loop structure from being a well-formed model (i.e. a XOR merge gateway followed, at some point after it, by a XOR split decision gateway). Example of that are gateways g3, g4 and g5, g8 in Fig. 1, where the decision gateways (g4 and g8) decides either to forward the process to later actions or to go back to the merge gateways (i.e. g3 and g5). In point (6), the function *preds* (i.e. object predecessors) returns a set of *connected* predecessor objects from an object to the start event, and the function *succs* (i.e. object successors) returns a set of *connected* successor objects from an object to the end event. The function names in the parenthesis are referring to the membership below.

In Fig. 2, an example of the difference between a well-structured and a well-formed model is illustrated. The model in (a) has a gateway with more than one input and output transitions and more than one start and end events at the same time, while the model in (b) satisfies the well-formedness requirements above. In order to automate the check of these conditions, we introduce equationally-defined predicate `(wfs)` for well-formed set of BPMN elements, checking whether the requirements above are satisfied for each set of elements.

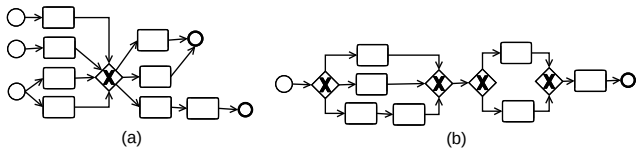


Fig. 2. (a)well-structured BPMN model. (b)well-formed BPMN Model.

```
subsort WFprocess < ObjectSet .
op wfs : Object ObjectSet ~> Bool .
ceq wfs(O,A) = true
 if wfstartend((O,A),noobject) /\ wfException((O,A),noobject) /\
    wfActivities((O,A),noobject) /\ wfGates((O,A),noobject) /\
    wfpath(O,(O,A),noobject) .
eq wfs(O,A) = false [owise] .
op <<_>> : ObjectSet -> ObjectSet [ctor].
cmb << O, A >> : WFprocess if wfs(O,A) .
```

The membership above performs a comparison between the set[4] which contains the well-formed objects of the same type and the set of the objects of that particular type in the process. If these two sets are identical, the condition is satisfied. Otherwise (`[owise]` function above), it returns *false*. On satisfying the well-formedness requirements, a process is a `WFprocess` (i.e. a subsort of the main sort `ObjectSet`). That is, an object can change its sort during execution if it satisfies certain conditions following the concept of Maude's conditional membership [14]. Having well-formed BPMN models allows us to formally define its behavioural semantics as detailed in the next section.

5 BPMN Behavioural Semantics

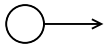
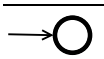
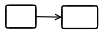
In this section two categories of semantics specification rules are defined. The first category contains the general rules for common patterns of behaviour in a BPMN model (subsection 5.1), while the second category contains more domain specific rules (subsection 5.2) which serve the introduced example in Section 2.

5.1 General Behavioural Rules

A `W-BPMN` process is *active* if one or more of its objects are active. Conversely, a `W-BPMN` process is *inactive* if all its objects are inactive. Hence, we introduce a boolean attribute `active` to indicate whether an object is active or not. As shown in Table 1, the start event is activated according to the rule `Initiate` which assigns the value *true* to the start event `active` attribute if there are no active objects in the process initial state. On the other side of the process, the rule `Terminate` terminates a process if it is active and the only active object is the end object using the function `isActive`. The rule rewrites the object set

[4] We use the operator `noobject` as the identity element for the `objectSet`

Table 1. BPMN Behavioural Semantics Rewriting Rules - I

BPMN – ——-> Rewrite Rules	
	`crl[Initiate]: CVcol*<<A,<E1:startEvent|active:false;AS1> >> => CVcol*<<A,<E1:startEvent|active:true;AS1> >> if isActive(<<A>>)=false.`
	`crl[Terminate]: CVcol * <<A,<E1:endEvent|active:true;AS1> >> => TerminatedSuccessfully if isActive (<< A >>) = false.`
	`rl[Seq]: << <X:K|out:tN1;active:true;AS1>, <Y:L|in:(tN1,T1);active:false;AS2>, A >> => << <X:K|out:tN1;active:false;AS1>, <Y:L|in:(tN1,T1);active:true;AS2>, A >>`
	`rl[ANDFork]: << <G1:aforkgate|out:T1;active:true;AS1>,A >> => << <G1:aforkgate|out:T1;active:false;AS1>, activateANDsuccessors(T1,A)>>`
	`rl[XORsplit]: (CVcol)*<< <G1:xsplitgate|out:T1;dF:T2;guard:GExp; error:tN2;controlValues:noCV;active:true;AS1>, A >> =>(CVcol)*<< <G1:xsplitgate|out:T1;dF:T2;guard:GExp; error:tN2;controlValues:assignCVs(GExp,CVs);active:false;AS1>, activateXORchild(T1,T2,tN2,GExp,assignCVs(GExp,CVs),A)>>`
	`rl[ORsplit]: (CVcol)*<< <G1:osplitgate|out:(tN1,T1);dF:tN2; guard:GExp;error:tN3;controlValues:noCV;active:true;AS1>,A>> => (CVcol)*<< <G1:osplitgate|out:(tN1,T1);error:tN3;dF:tN2; guard:GExp;controlValues:assignCVs(GExp,CVs);active:false;AS1>, (activateORchildren(evalORguard(GExp,assignCVs(GExp,CVs), notrans),A))>>`

into a descriptive statement indicating process termination (e.g. the term zero) as an indication of successful termination. For the *sequential* execution pattern, the rule `Seq` in Table 1 deactivates the first object `X` and activates the successor object `Y` which may be an activity or a gateway, where `K` and `L` are the objects CIDs, `tN1` is the transition that links the two objects, `AS1` and `AS2` are the remaining attributes that an object might have, and `A` is the other objects in the process. The *parallel* execution pattern (rule `ANDFork`): activates all immediate successor objects for the fork gateway concurrently. In the *exclusive decision-based* execution pattern (rule `XORsplit`): a decision must be taken based on the evaluation of guard conditions, which will allow the flow to go in one branch and prevent it from others (i.e., the expressions are mutually exclusive). While function `activateXORchild` activates the successor object, function `evalGuard` retrieves the transition linked to the successfully evaluated expression and then it can be used to connect with the object to be activated. The guard `Expression` is evaluated using the well-defined Maude modules (i.e. `NAT`, `STRING`, `BOOL`) for the possible guard operations mentioned in Section 3.

For *inclusive decision-based* (rule `ORsplit`): activates the OR-split successor(s) using function `activateORchildren` based on the evaluation of the guard expressions using function `evalORguard`, hence activating one *or more* successors. The function `evalORguard` takes the `Expression`(s), the control values and

the set of outgoing transitions from the OR-split gateway under consideration as inputs and retrieves the set of transitions whose associated guard expression evaluates to *true*. The definition below deals with the equality relation over the string values in guard expressions.

```
eq evalORguard(noexp,noCV,T1) = T1 .
eq evalORguard(((V1==S1, t N2) . GExp), ((V1:S1) .. CVs),T1)
 = evalORguard(GExp, CVs, (t N2, T1)) .
eq evalORguard(((V1==S1, t N2) . GExp), ((V1:S2) .. CVs),T1)
 = evalORguard(GExp,CVs,T1) [owise] .
```

In case of *joining parallel* activities, the `ajoingate` cannot be activated until all its predecessors have finished (i.e. deactivated). The (rule `ANDJoin`) in Table 2 activates the join gateway and the function `deactivatePreds` collects the immediate predecessors and deactivate them if active predecessors are found using function `activePreds`. As long as the XOR activates only one branch, then the *merging* behaviour of the exclusive gateways activates an `xmergegate` object if it has one active immediate predecessor, and it deactivates this predecessor using (rule `XORmerge`) in Table 2.

In case of OR merge semantics, the number of activated branches that it should wait for before activating the successor object is unknown to the merge gateway. This situation may result in lack of synchronization [15] where the merge gateway succeeding object is activated more than once. In order to overcome this situation, the attribute `itsSplit` is introduced to relate a merge gateway to its split gateway, in order to keep track of the number of activated flows that need to be deactivated. Moreover, function `ready2Merge` checks if all the activated immediate predecessors of the OR merge gateway are active and ready for the merge. Here, the relation between the two gateways as one block (check Section 4) facilitates the efficient evaluation and execution of the guards. It enables the merge gateway to know how many branches have been activated after its corresponding OR-split gateway. In (rule `ORmerge`) in Table 2, Boolean function `ready2Merge` is used to take the activated transitions resulted from the evaluation function `evalORguard` and checks if the same number of predecessors is active and ready to be merged.

```
eq Ready2Merge(notrans,T1,A)=true .
eq Ready2Merge((tN1,T1),(tN2,T2),(<X:K|out:(tN2,T3);active:true;AS1>,A))
 = Ready2Merge((T1),(T2),(<X:K|out:(tN2,T3);active:true;AS1>,A)) [owise].
```

The function `assignCVs` is defined to enable the automatic assignment of control values to the corresponding gateways at run time. It takes the given collection of control values and assigns them to their corresponding split gateway object attribute `controlValues` in the process. In particular, it matches the guard expression variable name and the control value variable name, then retrieves the set of control values for the split gateway. It allows the modeller to enter the control values as a collection as part of the initial configuration of the process without modifying the process itself. The result of the process execution is of the form defined as (`CVcollection * ObjectSet -> Trace`). The resulting trace is a possible trace for the model. A trace is a possible

Table 2. BPMN Behavioural Semantics Rewriting Rules - II

BPMN – ——-> Rewrite Rules	
	crl[ANDJoin]:<< <G1:ajoingate | in:T1;itsSplit:G2; active:false;ToBActive:true;AS1>,<G2:aforkgate|AS3>,A>> => <<<G1:ajoingate|in:T1;itsSplit:G2; active:true;ToBActive:false;AS1>,<G2:aforkgate|AS3> ,A>> if activePreds(T1,preds1(<G2:aforkgate|AS3>, <G1:ajoingate|in:T1;itsSplit:G2;active:true; ToBActive:false;AS1>,(A,<G2:aforkgate|AS3>),nobject,nobject)) == false∧isActive(<<imPreds(T1,A,noobject)>>) == false.
	crl[XORmerge]:<< <X:K|out:(tN1,T2); active:true; AS1>, <G1:xmergegate | in:(tN1,T1); active:false; AS2 >,A>> => << <X:K|out:(tN1,T2); active:false; AS1>,<G1:xmergegate| in:(tN1,T1); active:true; AS2>,A>> if K =/= xsplitgate
	crl[ORmerge]: << <G1:omergegate|in:(tN1,T1);itsSplit:G2; active:false;AS1>,<G2:osplitgate|active:false;guard:GExp; controlValues:CVs;AS2>,A>> => <<<G1:omergegate|in:(tN1,T1);itsSplit:G2;active:true;AS1>, <G2:osplitgate|active:false;guard:GExp;controlValues:CVs;AS2>, deactivateOPreds((tN1,T1),A)>> if Ready2Merge(evalORguard(GExp,CVs,notrans),(tN1,T1),A)

execution of the BPMN model, and O_{Traces} is the set of all possible traces for executing a process. If the model does not have diverging elements (e.g. gateways), then it will have one possible trace of execution; i.e. $\forall o_i \in OS$ $((Cid(o_i) \notin G \wedge eventType(o_i) \neq \texttt{exception}) \rightarrow |O_{Traces}| = 1)$. However, if the model contains diverging elements, then there will be more than one possible trace of the BPMN model; i.e. $\exists o_i \in OS$ $((Cid(o_i) \in \{\texttt{ANDfork}, \texttt{XORsplit}, \texttt{ORsplit}, \texttt{exception}\}) \rightarrow |O_{Traces}| > 1)$. Thus we can consider the possible executable traces for the model if we listed the set of the possible collections of control values for a certain model. In this case, by simulating the process execution, it will result in the set of the possible traces for the model with respect to gateway routing. For example, a possible trace from the example in Fig. 1 can be represented as: `(((Authorized?:"YES"), (AuthorizedCR?:"YES"), (MoreCR1?:"NO"), (MoreCR2? : "NO")) * ReleaseBaseline)`.

5.2 Domain-Specific Semantics

Each BP represents a specific work procedure carrying its characteristics, conditions and constraints. This can be reflected by specific patterns of behaviour, assigned to process elements, which are dependent on the corresponding business environment. In Fig. 1, task `Retrieve CI` is active only if a (`CR`) is open and data object `Baseline` status is changed to *released* if the task `Release Baseline` is active. Such conditional behaviours are modelled graphically for presentation purposes in Fig. 3 (e) and (g) respectively, while the corresponding term rewrite rules are coded in Maude using the syntactic notation presented

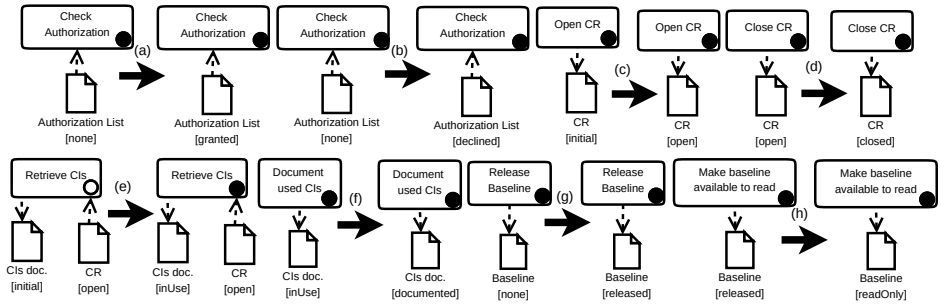


Fig. 3. Domain Context Rules for Release Baselines Example

in Section 3. In Fig. 3 parts (a)-(h), the possible status (defining its life cycle) for a Change Request `CR` data object are: *initial*, *open* and *closed*; for a Configuration Item `CI doc` are: *initial*, *inUse* and *documented*; for `Authorization List` are: *none*, *granted* and *declined* and for a `Baseline` are: *none*, *released* and *readOnly*. Some rules are applied when the corresponding task is active (marked by the black circle in the corner of the task) such as `Check Authorization` and others can change their inactive state to an active state depending on the connected data object's status (e.g. `Retrieve CIs`). In the example, an authorization is checked, if the requester is listed in the `Authorization List` then the data object status is changed to *granted* (rule (a) in Fig. 3), otherwise, the status will change to declined (rule (b)). Similarly, for each `CR`, it is opened (rule (c)), a relevant `CI doc` is called to be changed (rule (e)) and the `CR` is closed (rule (d)). The `CI doc` is set to be `inUse` (rule (e)) and then to be `documented` (rule (f)). Finally, a `Baseline` is `released` when task *Release Baseline* is active (rule (g)) then made as a `readOnly` document (rule (h)). In the next section we introduce how the specified syntax and semantics are used to verify BPMN processes and use this as the basis for checking a Configuration Management process compliance.

6 Verifying BP models

Our proposed approach enables the formal analysis of the BPMN models at design time using Maude's LTL model checker [8]. The BP model is checked against some defined properties to see if they are satisfied by the model or not. Maude's LTL Model Checker associates a Kripke structure ($\mathcal{K}$) with the rewrite theory ($\mathcal{R}$) which represents a model M. Then, the model checker solves a satisfaction problem of the form $\mathcal{K}(\mathcal{R}, k)_{\Pi}, [t] \models \varphi$ where k is a set of initial states from the rewrite theory $\mathcal{R}$, Π defines the state predicates to be considered, $[t]$ a kind of initial states, and φ is the property to be checked. The result of model checking is *true*, i.e. the property is satisfied, or a possible trace of the model in which the property does not hold; i.e. counterexample. The example BP model in Fig. 1 represents a subprocess of the CM process in a software company. This acts

Table 3. CMMI-CM Requirements and the corresponding LTL formulae

	Textual Requirements, LTL Formulae and MC Results
Req.1	Obtain authorization before releasing baselines (1.3.1)
LTL	`[](~executed("Release Baseline") W conditionGuard(Authorized?:"YES"))`
MC	`result Bool: true`
Req.2	Release baselines of the documented CIs (1.3.2)
LTL	`[](status("CI doc", documented) → <>executed("Release Baseline"))`
MC	`result Bool: true`
Req.3	Track the status of change requests to closure (2.1.5)
LTL	`[]((status("CR-1", open) →<> status("CR-1", closed)) ∧ (status("CR-2", open) →<> status("CR-2", closed)) ∧ (status("CR-3", open) →<> status("CR-3", closed)))`
MC	`result Bool: true`

as the model specifications which are specified by the rewrite theory $\mathcal{R}$ earlier in Section 3 and Section 5, while the property specifications φ are modelled from the CMMI [13] best practices. The set of predicates Π we use are: (1) `executed`: determines the activation of an activity, (2) `status`: determines the status change of a data object, and (3) `conditionGuard`: determines the control value for a gateway guard. The requirement specifications (φ) are modelled in LTL formulae as described in Table 3[5]. The goal is to check if the model satisfies properties in (φ) by means of $(\mathcal{K}(\mathcal{R}, k)_{\Pi}, [t] \models \varphi)$ using Maude's LTL model checker. The formula: `reduce in` $\langle ModuleName \rangle$ `: modelCheck`$(\langle InitialState \rangle, \langle LTLproperty \rangle)$ is used. The command `modelCheck` takes the initial state of the system (`initial`), where the process is *inactive* and the LTL formula for the property (as in Table 3).

Req-1: *Obtain authorization before releasing baselines.* Depending on the value of variable `Authorized?`, the flow can proceed to activity `ReleaseBaseline` in case it is authorized or to gateway `g10` without releasing the baseline (cf. rules (a) and (b) in Fig. 3). The LTL symbol "weak until" (`a W b`) used in the formula in Table 3 requires that `a` remains *true* until `b` becomes *true*, but does not require that `b` ever does become *true* (i.e. `a` may remain *true* forever). Based on the data object `Authorization List` value, which is `"YES"` in the example, the activity `Release Baseline` can be activated. Otherwise, if the value is `"NO"`, the activity may remain not active forever. One advantage of the LTL is the ability to express safety properties of this kind.

Req-2: *Release baselines of the documented CIs.* Whenever the CI is documented, then baseline should be released at some point in the future. The LTL formula in the table indicates that the CI document has the status `documented`, which *implies* that activity `Release Baseline` is eventually (i.e. LTL symbol

[5] The numbers in the parentheses are references to the source in [13], for example, (1.2.3) refers to the first specific goal, second specific practice, and third sub-practice in CM Process Area.

<>) executed. By checking the model, it retrieves *true* which means the property is already satisfied by the model.

Req-3: *Track the status of change requests to closure*. It checks that all change requests are resolved before releasing a baseline. The LTL formula specifies that whenever a change request `CR` is *open*, it is eventually *closed*. Note that we are considering three change requests in the example. The check returns *true*.

The CMMI requires a CM process to satisfy a number of best practices (some of them are modelled as requirements in Table 3) in order to be compliant with the CMMI standard model along with other CMMI process areas. Hence the requirements for releasing a baseline should be satisfied by the BP model. If we define the requirements as $\varphi = \{\phi_1, \phi_2, \phi_3\}$, then the compliance check result is dependent on the relation $\mathcal{K}(\mathcal{R}, k)_{\Pi}, [t] \models \varphi$. Based on the above model checking results, the satisfaction relation above will result in *true*. This means the BP in Fig. 1 is *compliant* with the CMMI CM process area, assuming it contains only the used properties in Table 3.

7 Related Work and Conclusions

There are many attempts to transform the BPMN as a graphical language into Petri nets [3], YAWL [6], graphs [4] and CSP [5]. A mapping from a subset of BPMN into Petri nets was proposed in [3] reserving the classical Petri nets limitation in representing the inclusive OR gateway. Another mapping into YAWL was introduced in [6] following a well-formed models notion and lacking the data objects and inclusive OR gateway semantics. Graph rewrite rules were used in [4] for conformance checking and visualization purposes without using a well-formedness notion or handling of data objects. In [5], CSP was used to define the BPMN semantics, followed by a refinement procedure for property checking. The approach in [9] used Maude to formally represent workflow patterns however, the notion of well-formedness is not used and the data objects are not formalized. To the best of our knowledge, no approach has included, as part of the language formalization, a formal representation with evaluation semantics for the conditions guarding the BPMN gateways before this work. Moreover, our approach provide a comprehensive formalization for the inclusive OR merge gateways that prevent lack of synchronization. We believe it remains a problem in many approaches (e.g. [3, 9, 6]). As for compliance problem, in [16], based on model-driven development, a metamodel compliance checking was implemented in an object-oriented hypertext representation of requirements tool RETH. A semi-automatic framework for managing compliance throughout the BP lifecycle based on defined compliance patterns was introduced in [17]. In [18], BPMN-Q, a BPMN based query language, was used for property specifications.

In this paper we have presented the formal syntax and behavioural semantics specifications for a subset of the BPMN. We have solved some of BPMN challenging issues related to its ambiguity and process structure, conditional expressions, and handling of data objects. A set of domain-specific rules are introduced for simulating the behaviour of data objects. Moreover, we introduce a

well-formed semantics for the merge gateways by using the block structure. The resulting BPMN models are verified using Maude verification toolkit to ensure well-formedness. An illustrated example is used to prove the applicability of the formalization in compliance checking for BPs like the CM. In future work, the approach may be extended to model more BPMN elements, such as the event-based gateways, process collaboration and exception handling. A further study of the soundness of the resulting models is undergoing. The current prototype provides the formalization of BPs in Maude and we are planning its integration within a modelling environment (e.g. Eclipse) to map BPMN models into terms in Maude in order to verify them with Maude's verification toolkit.

References

[1] OMG: BPMN 2.0. Technical Report formal/2011-01-03, OMG (2011)
[2] Börger, E.: Approaches to Modeling BPs: a Critical Analysis of BPMN, Workflow Patterns and YAWL. Softw. Syst. Model. **11**(3) (2012) 305–318
[3] Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and Analysis of Business Process Models in BPMN. Inf. Softw. Technol. **50**(12) (2008) 1281–1294
[4] Gorp, P.V., Dijkman, R.M.: A Visual token-based Formalization of BPMN 2.0 based on in-place Transformations. Infor. & Soft. Tech. **55**(2) (2013) 365–394
[5] Wong, P.Y., Gibbons, J.: Formalisations and Applications of BPMN. Sci. Comput. Program. **76**(8) (2011) 633–650
[6] Ye, J., Song, W.: Transformation of BPMN Diagrams to YAWL Nets. Journal of Software **5**(4) (2010)
[7] Muehlen, M.Z., Recker, J.: How Much Language Is Enough? Theoretical and Practical Use of the Business Process Modeling Notation. In: Advanced Information Systems Engineering. CAiSE '08, Berlin, Springer-Verlag (2008) 465–479
[8] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - a High-Performance Logical Framework. Springer (2007)
[9] Grande, L.H.: Introducción a la notación BPMN y su relación con las estrategias del lenguaje Maude. Master's thesis, Facultad de Informática (2009)
[10] Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL Model Checker. In: Proceedings of WRLA02. Volume 71 of ENTCS., Elsevier (2002)
[11] Meseguer, J.: Conditional Rewriting Logic As a Unified Model of Concurrency. Theoretical Computer Science **96**(1) (April 1992) 73–155
[12] Boronat, A., Meseguer, J.: An Algebraic Semantics for MOF. In Fiadeiro, J.L., Inverardi, P., eds.: FASE. LNCS, Springer (2008) 377–391
[13] SEI: CMMI. Technical Report CMU/SEI-2010-TR-033, CMU-SEI (2011)
[14] Meseguer, J.: Membership Algebra as a Logical Framework for Equational Specification. In Presicce, F.P., ed.: Recent Trends in Algebraic Development Techniques. Volume 1376 of LNCS., London, Springer-Verlag (1998) 18–61
[15] van der Aalst, W., Hirnschall, A., Verbeek, H.: An Alternative Way to Analyze Workflow Graphs. CAiSE'02, London, UK, Springer-Verlag (2002) 535–552
[16] Kaindl, H., Kramer, S., Hailing, M., Harput, V.: Metamodel-Compliance Checking of Requirements in a Semiformal Representation. In: CAiSE. Volume 74. (2003)
[17] Türetken, O., Elgammal, A., van den Heuvel, W.J., Papazoglou, M.P.: Enforcing Compliance on Business Processes through the use of Patterns. In: ECIS. (2011)
[18] Awad, A.: A Compliance Management Framework for Business Process Models. PhD thesis, Hasso-Plattner-Institute, Potsdam, Germany (May 2010)

Towards Static Analysis of Functional Programs using Tree Automata Completion

Thomas Genet

INRIA/IRISA, Université de Rennes, France
genet@irisa.fr

Abstract. This paper presents the first step of a wider research effort to apply tree automata completion to the static analysis of functional programs. Tree Automata Completion is a family of techniques for computing or approximating the set of terms reachable by a rewriting relation. The completion algorithm we focus on is parameterized by a set E of equations controlling the precision of the approximation and influencing its termination. For completion to be used as a static analysis, the first step is to guarantee its termination. In this work, we thus give a sufficient condition on E and $\mathcal{T}(\mathcal{F})$ for completion algorithm to always terminate. In the particular setting of functional programs, this condition can be relaxed into a condition on E and $\mathcal{T}(\mathcal{C})$ (terms built on the set of constructors) that is closer to what is done in the field of static analysis, where abstractions are performed on data.

1 Introduction

Computing or approximating the set of terms reachable by rewriting has more and more applications. For a Term Rewriting System (TRS) $\mathcal{R}$ and a set of terms $L_0 \subseteq \mathcal{T}(\mathcal{F})$, the set of reachable terms is $\mathcal{R}^*(L_0) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in L_0, s \to_{\mathcal{R}}{}^* t\}$. This set can be computed exactly for specific classes of $\mathcal{R}$ [10] but, in general, it has to be approximated. Applications of the approximation of $\mathcal{R}^*(L_0)$ are ranging from cryptographic protocol verification [1], to static analysis of various programming languages [5] or to TRS termination proofs [15]. Most of the techniques compute such approximations using tree automata as the core formalism to represent or approximate the (possibly) infinite set of terms $\mathcal{R}^*(L_0)$. Most of them also rely on a Knuth-Bendix completion-like algorithm completing a tree automaton $\mathcal{A}$ recognizing L_0 into an automaton $\mathcal{A}^*$ recognizing exactly, or over-approximating, the set $\mathcal{R}^*(L_0)$. As a result, these techniques can be refered as *tree automata completion* techniques [9, 22, 8, 4, 13, 19]. A strength of this algorithm, and at the same time a weakness, is that its precision is parameterized by a function [8] or a set of equations [13]. It is a strength because tuning the approximation function (or equations) permits to adapt the precision of completion to a specific goal to tackle. This is what made it successful for program and protocol verification. On the other hand, this is a weakness because it is difficult to guarantee its termination.

In this paper, we define a simple sufficient condition on the set of equations for the tree automata completion algorithm to terminate. This condition, which is strong in general, reveals to be natural and well adapted for the approximation of reachable terms when TRSs encode typed functional programs. We thus obtain a way to automatically over-approximate the set of all reachable program states of a functional program, or even restrict it to the set of all results. Thus we can over-approximate the image of a functional program.

2 Related work

Tree automata completion. With regards to most papers about completion [9, 22, 8, 4, 13, 19], our contribution is to give the first criterion *on the approximation* for the completion to terminate. Note that it is possible to guarantee termination of the completion by inferring an approximation adapted to the TRS under concern, like in [20]. In this case, given a TRS, the approximation is fixed and unique. Our solution is more flexible because it lets the user change the precision of the approximation while keeping the termination guarantee. In [22], T. Takai have a completion parameterized by a set of equations. He also gives a termination proof for its completion but only for some restricted classes of TRSs. Here our termination proof holds for any left-linear TRS provided that the set of equations satisfy some properties.

Static analysis of functional programs. With regards to static analysis of functional programs using grammars or automata, our contribution is in the scope of data-flow analysis techniques, rather than control-flow analysis. More precisely, we are interested here in predicting the results of a function [21], rather than predicting the control flow [18]. Those two papers, as well as many other ones, deal with higher order functions using complex higher-order grammar formalisms (PMRS and HORS). Higher-order functions are not in the scope of the solution we propose here. However, we obtained some preliminary results suggesting that an extension to higher order functions is possible and gives relevant results (see Section 6). Furthermore, using equations, approximations are defined in a more declarative and flexible way than in [21], where they are defined by a dedicated algorithm. Besides, the verification mechanisms of [21] use automatic abstraction refinement. This can be also performed in the completion setting [3] and adapted to the analysis of functional programs [14]. Finally, using a simpler (first order) formalism, *i.e.* tree automata, makes it easier to take into account some other aspects like: evaluation strategies and built-ins types (see Section 6) that are not considered by those papers.

3 Background

In this section, we introduce some definitions and concepts that will be used throughout the rest of the paper (see also [2, 7]). Let $\mathcal{F}$ be a finite set of symbols, each associated with an arity function, and let $\mathcal{X}$ be a countable set of *variables*.

$\mathcal{T}(\mathcal{F},\mathcal{X})$ denotes the set of *terms* and $\mathcal{T}(\mathcal{F})$ denotes the set of *ground terms* (terms without variables). The set of variables of a term t is denoted by $\mathcal{V}ar(t)$. A *substitution* is a function σ from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F},\mathcal{X})$, which can be uniquely extended to an endomorphism of $\mathcal{T}(\mathcal{F},\mathcal{X})$. A *position* p for a term t is a finite word over $\mathbb{N}$. The empty sequence λ denotes the top-most position. The set $\mathcal{P}os(t)$ of positions of a term t is inductively defined by $\mathcal{P}os(t) = \{\lambda\}$ if $t \in \mathcal{X}$ or t is a constant and $\mathcal{P}os(f(t_1,\ldots,t_n)) = \{\lambda\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathcal{P}os(t_i)\}$ otherwise. If $p \in \mathcal{P}os(t)$, then $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the term obtained by replacement of the subterm $t|_p$ at position p by the term s.

A *term rewriting system* (TRS) $\mathcal{R}$ is a set of *rewrite rules* $l \to r$, where $l, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. A rewrite rule $l \to r$ is *left-linear* if each variable of l occurs only once in l. A TRS $\mathcal{R}$ is left-linear if every rewrite rule $l \to r$ of $\mathcal{R}$ is left-linear. The TRS $\mathcal{R}$ induces a rewriting relation $\to_{\mathcal{R}}$ on terms as follows. Let $s, t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and $l \to r \in \mathcal{R}$, $s \to_{\mathcal{R}} t$ denotes that there exists a position $p \in \mathcal{P}os(s)$ and a substitution σ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$. Given a TRS $\mathcal{R}$, $\mathcal{F}$ can be split into two disjoint sets $\mathcal{C}$ and $\mathcal{D}$. All symbols occurring at the root position of left-hand sides of rules of $\mathcal{R}$ are in $\mathcal{D}$. $\mathcal{D}$ is the set of defined symbols of $\mathcal{R}$, $\mathcal{C}$ is the set of constructors. Terms in $\mathcal{T}(\mathcal{C})$ are called *data-terms*. The reflexive transitive closure of $\to_{\mathcal{R}}$ is denoted by $\to^*_{\mathcal{R}}$ and $s \to^!_{\mathcal{R}} t$ denotes that $s \to^*_{\mathcal{R}} t$ and t is irreducible by $\mathcal{R}$. The set of irreducible terms w.r.t. a TRS $\mathcal{R}$ is denoted by $\textsc{Irr}(\mathcal{R})$. The set of $\mathcal{R}$-descendants of a set of ground terms I is $\mathcal{R}^*(I) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in I \text{ s.t. } s \to^*_{\mathcal{R}} t\}$. A TRS $\mathcal{R}$ is sufficiently complete if for all $s \in \mathcal{T}(\mathcal{F})$, $(R^*(\{s\}) \cap \mathcal{T}(\mathcal{C})) \neq \emptyset$.

An *equation set* E is a set of *equations* $l = r$, where $l, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$. The relation $=_E$ is the smallest congruence such that for all substitution σ we have $l\sigma =_E r\sigma$. Given a TRS $\mathcal{R}$ and a set of equations E, a term $s \in \mathcal{T}(\mathcal{F})$ is rewritten modulo E into $t \in \mathcal{T}(\mathcal{F})$, denoted $s \to_{\mathcal{R}/E} t$, if there exist $s' \in \mathcal{T}(\mathcal{F})$ and $t' \in \mathcal{T}(\mathcal{F})$ such that $s =_E s' \to_{\mathcal{R}} t' =_E t$. The reflexive transitive closure $\to^*_{\mathcal{R}/E}$ of $\to_{\mathcal{R}/E}$ is defined as usual except that reflexivity is extended to terms equal modulo E, *i.e.* for all $s, t \in \mathcal{T}(\mathcal{F})$ if $s =_E t$ then $s \to^*_{\mathcal{R}/E} t$. The set of $\mathcal{R}$-descendants modulo E of a set of ground terms I is $\mathcal{R}^*_E(I) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in I \text{ s.t. } s \to^*_{\mathcal{R}/E} t\}$.

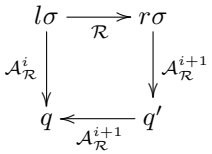
Let $\mathcal{Q}$ be a countably infinite set of symbols with arity 0, called *states*, such that $\mathcal{Q} \cap \mathcal{F} = \emptyset$. $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ is called the set of *configurations*. A *transition* is a rewrite rule $c \to q$, where c is a configuration and q is state. A transition is *normalized* when $c = f(q_1,\ldots,q_n)$, $f \in \mathcal{F}$ is of arity n, and $q_1,\ldots,q_n \in \mathcal{Q}$. An *$\epsilon$-transition* is a transition of the form $q \to q'$ where q and q' are states. A bottom-up non-deterministic finite tree automaton (*tree automaton* for short) over the alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_F, \Delta \rangle$, where $\mathcal{Q}_F$ is a finite subset of $\mathcal{Q}$, Δ is a finite set of normalized transitions and ϵ-transitions. The transitive and reflexive *rewriting relation* on $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ induced by the set of transitions Δ (resp. all transitions except ϵ-transitions) is denoted by $\to^*_{\Delta}$ (resp. $\to^{\not\epsilon *}_{\Delta}$). When Δ is attached to a tree automaton $\mathcal{A}$ we also note those two relations $\to_{\mathcal{A}}{}^*$ and $\to^{\not\epsilon *}_{\mathcal{A}}$, respectively. A tree automaton $\mathcal{A}$ is complete if for all $s \in \mathcal{T}(\mathcal{F})$ there exists

a state q of $\mathcal{A}$ such that $s \rightarrow_{\mathcal{A}}{}^{*} q$. The language (resp. $\not$language) recognized by $\mathcal{A}$ in a state q is $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \rightarrow_{\mathcal{A}}^{*} q\}$ (resp. $\mathcal{L}^{\not}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \rightarrow_{\mathcal{A}}^{\not *} q\}$). A state q of an automaton $\mathcal{A}$ is *reachable* (resp. $\not$reachable) if $\mathcal{L}(\mathcal{A}, q) \neq \emptyset$ (resp. $\mathcal{L}^{\not}(\mathcal{A}, q) \neq \emptyset$). We define $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in \mathcal{Q}_F} \mathcal{L}(\mathcal{A}, q)$. A set of transitions Δ is $\not$deterministic if there are no two normalized transitions in Δ with the same left-hand side. A tree automaton $\mathcal{A}$ is $\not$deterministic if its set of transitions is $\not$deterministic. Note that if $\mathcal{A}$ is $\not$deterministic then for all states q_1, q_2 of $\mathcal{A}$ such that $q_1 \neq q_2$, we have $\mathcal{L}^{\not}(\mathcal{A}, q_1) \cap \mathcal{L}^{\not}(\mathcal{A}, q_2) = \emptyset$.

4 Tree Automata Completion Algorithm

Tree Automata Completion algorithms were proposed in [16, 9, 22, 13]. They are very similar to a Knuth-Bendix completion except that they run on two distinct sets of rules: a TRS $\mathcal{R}$ and a set of transitions Δ of a tree automaton $\mathcal{A}$.

Starting from a tree automaton $\mathcal{A}_0 = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_0 \rangle$ and a left-linear TRS $\mathcal{R}$, the algorithm computes a tree automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \mathcal{R}^{*}(\mathcal{L}(\mathcal{A}_0))$ or $\mathcal{L}(\mathcal{A}') \supseteq \mathcal{R}^{*}(\mathcal{L}(\mathcal{A}_0))$. The algorithm iteratively computes tree automata $\mathcal{A}^1_{\mathcal{R}}$, $\mathcal{A}^2_{\mathcal{R}}$, ... such that $\forall i \geq 0 : \mathcal{L}(\mathcal{A}^i_{\mathcal{R}}) \subseteq \mathcal{L}(\mathcal{A}^{i+1}_{\mathcal{R}})$ until we get an automaton $\mathcal{A}^k_{\mathcal{R}}$ with $k \in \mathbb{N}$ and $\mathcal{L}(\mathcal{A}^k_{\mathcal{R}}) = \mathcal{L}(\mathcal{A}^{k+1}_{\mathcal{R}})$. For all $i \in \mathbb{N}$, if $s \in \mathcal{L}(\mathcal{A}^i_{\mathcal{R}})$ and $s \rightarrow_{\mathcal{R}} t$, then $t \in \mathcal{L}(\mathcal{A}^{i+1}_{\mathcal{R}})$. Thus, if $\mathcal{A}^k_{\mathcal{R}}$ is a fixpoint then it also verifies $\mathcal{L}(\mathcal{A}^k_{\mathcal{R}}) \supseteq \mathcal{R}^{*}(\mathcal{L}(\mathcal{A}_0))$. To construct $\mathcal{A}^{i+1}_{\mathcal{R}}$ from $\mathcal{A}^i_{\mathcal{R}}$, we achieve a *completion step* which consists in finding *critical pairs* between $\rightarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{A}^i_{\mathcal{R}}}$. A critical pair is a triple $(l \rightarrow r, \sigma, q)$ where $l \rightarrow r \in \mathcal{R}$, $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and $q \in \mathcal{Q}$ such that $l\sigma \rightarrow^{*}_{\mathcal{A}^i_{\mathcal{R}}} q$ and $r\sigma \not\rightarrow^{*}_{\mathcal{A}^i_{\mathcal{R}}} q$. For $r\sigma$ to be recognized by the same state and thus model the rewriting of $l\sigma$ into $r\sigma$, it is enough to add the necessary transitions to $\mathcal{A}^i_{\mathcal{R}}$ to obtain $\mathcal{A}^{i+1}_{\mathcal{R}}$ such that $r\sigma \rightarrow^{*}_{\mathcal{A}^{i+1}_{\mathcal{R}}} q$. In [22, 13], critical pairs are joined in the following way:



From an algorithmic point of view, there remains two problems to solve: find all the critical pairs $(l \rightarrow r, \sigma, q)$ and find the transitions to add to $\mathcal{A}^i_{\mathcal{R}}$ to have $r\sigma \rightarrow^{*}_{\mathcal{A}^{i+1}_{\mathcal{R}}} q$. The first problem, called matching, can be efficiently solved using a specific algorithm [8, 10]. The second problem is solved using Normalization.

4.1 Normalization

The normalization function replaces subterms either by states of $\mathcal{Q}$ (using transitions of Δ) or by new states. A state q of $\mathcal{Q}$ is used to normalize a term t if $t \rightarrow^{\not}_{\Delta} q$. Normalizing by reusing states of $\mathcal{Q}$ and transitions of Δ permits to preserve the $\not$determinism of $\rightarrow^{\not}_{\Delta}$. Indeed, $\rightarrow^{\not}_{\Delta}$ can be kept deterministic during completion though $\rightarrow_{\Delta}$ cannot.

Definition 1 (New state). *Given a set of transitions Δ, a new state (for Δ) is a state of $\mathcal{Q} \setminus \mathcal{Q}_f$ not occurring in left or right-hand sides of rules of Δ* [1].

We here define normalization as a bottom-up process. This definition is simpler and equivalent to top-down definitions [13]. In the recursive call, the choice of the context $C[\,]$ may be non deterministic but all the possible results are the equivalent modulo state renaming.

Definition 2 (Normalization). *Let Δ be a set of transitions defined on a set of states $\mathcal{Q}$, $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \setminus \mathcal{Q}$. Let $C[\,]$ be a non empty context of $\mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \setminus \mathcal{Q}$, $f \in \mathcal{F}$ of arity n, and $q, q', q_1, \ldots, q_n \in \mathcal{Q}$. The normalization function is inductively defined by:*

1. $Norm_\Delta(f(q_1, \ldots, q_n) \to q) = \{f(q_1, \ldots, q_n) \to q\}$
2. $Norm_\Delta(C[f(q_1, \ldots, q_n)] \to q) = \{f(q_1, \ldots, q_n) \to q'\} \cup Norm_{\Delta \cup \{f(q_1,\ldots,q_n) \to q'\}}(C[q'] \to q)$

   where either $(f(q_1, \ldots, q_n) \to q' \in \Delta)$ or (q' is a new state for Δ and $\forall q'' \in Q : f(q_1, \ldots, q_n) \to q'' \notin \Delta$).

In the second case of the definition, if there are several states q' such that $f(q_1, \ldots, q_n) \to q' \in \Delta$, we arbitrarily choose one of them. We illustrate the above definition on the normalization of a simple transition.

Example 1. Given $\Delta = \{b \to q_0\}$, $Norm_\Delta(f(g(a), b, g(a)) \to q) = \{a \to q_1, g(q_1) \to q_2, b \to q_0, f(q_2, q_0, q_2) \to q\}$

4.2 One step of completion

A step of completion only consists in joining critical pairs. We first need to formally define the substitutions under concern: *state substitutions*.

Definition 3 (State substitutions, $\Sigma(\mathcal{Q}, \mathcal{X})$). *A* state substitution *over an automaton $\mathcal{A}$ with a set of states $\mathcal{Q}$ is a function $\sigma : \mathcal{X} \mapsto \mathcal{Q}$. We can extend this definition to a morphism $\sigma : \mathcal{T}(\mathcal{F}, \mathcal{X}) \mapsto \mathcal{T}(\mathcal{F}, \mathcal{Q})$. We denote by $\Sigma(\mathcal{Q}, \mathcal{X})$ the set of state substitutions built over $\mathcal{Q}$ and $\mathcal{X}$.*

Definition 4 (Set of critical pairs). *Let a TRS $\mathcal{R}$ and a tree automaton $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$. The set of critical pairs between $\mathcal{R}$ and $\mathcal{A}$ is $CP(\mathcal{R}, \mathcal{A}) = \{(l \to r, \sigma, q) \mid l \to r \in \mathcal{R},\ q \in \mathcal{Q},\ \sigma \in \Sigma(\mathcal{Q}, \mathcal{X}),\ l\sigma \to^*_\mathcal{A} q,\ r\sigma \not\to^*_\mathcal{A} q\}$.*

Recall that the completion process builds a sequence $\mathcal{A}^0_\mathcal{R}, \mathcal{A}^1_\mathcal{R}, \ldots, \mathcal{A}^k_\mathcal{R}$ of automata such that if $s \in \mathcal{L}(\mathcal{A}^i_\mathcal{R})$ and $s \to_\mathcal{R} t$ then $t \in \mathcal{L}(\mathcal{A}^{i+1}_\mathcal{R})$. One step of completion, *i.e.* the process computing $\mathcal{A}^{i+1}_\mathcal{R}$ from $\mathcal{A}^i_\mathcal{R}$, is defined as follows. Again, the following definition is a simplification of the definition of [13].

[1] Since $\mathcal{Q}$ is a countably infinite set of states, $\mathcal{Q}_f$ and Δ are finite, a new state can always be found.

Definition 5 (One step of completion). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, $\mathcal{R}$ be a left-linear TRS. The one step completed automaton is $\mathcal{C}_\mathcal{R}(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, Join^{CP(\mathcal{R},\mathcal{A})}(\Delta) \rangle$ where $Join^S(\Delta)$ is inductively defined by:*

- $Join^\emptyset(\Delta) = \Delta$
- $Join^{\{(l \to r, q, \sigma)\} \cup S}(\Delta) = Join^S(\Delta \cup \Delta')$ *where*
  $\Delta' = \{q' \to q\}$ *if there exists* $q' \in \mathcal{Q}$ *s.t.* $r\sigma \to_\Delta^{\not\epsilon *} q'$, *and otherwise*
  $\Delta' = Norm_\Delta(r\sigma \to q') \cup \{q' \to q\}$ *where* q' *is a new state for* Δ

Example 2. Let $\mathcal{A}$ be a tree automaton with $\Delta = \{f(q_1) \to q_0, a \to q_1, g(q_1) \to q_2\}$. If $\mathcal{R} = \{f(x) \to f(g(x))\}$ then $CP(\mathcal{R}, \mathcal{A}) = \{(f(x) \to f(g(x)), \sigma_3, q_0)\}$ with $\sigma_3 = \{x \mapsto q_1\}$, because $f(x)\sigma_3 \to_{\mathcal{A}^*} q_0$ and $f(x)\sigma_3 \to_\mathcal{R} f(g(x))\sigma_3$. We have $f(g(x))\sigma_3 = f(g(q_1))$ and there exists no state q such that $f(g(q_1)) \to_\mathcal{A}^{\not\epsilon *} q$. Hence, $Join^{\{(f(x) \to f(g(x)), \sigma_3, q_0)\}}(\Delta) = Join^\emptyset(\Delta \cup Norm_\Delta(f(g(q_1)) \to q_3) \cup \{q_3 \to q_0\})$. Since $Norm_\Delta(f(g(q_1)) \to q_3) = \{f(q_2) \to q_3, q(q_1) \to q_2\}$, we get that $\mathcal{C}_\mathcal{R}(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q} \cup \{q_3\}, \mathcal{Q}_f, \Delta \cup \{f(q_2) \to q_3, q_3 \to q_0\} \rangle$.

4.3 Simplification of Tree Automata by Equations

In this section, we define the *simplification* of tree automata $\mathcal{A}$ w.r.t. a set of equations E. This operation permits to over-approximate languages that cannot be recognized *exactly* using tree automata completion, *e.g.* non regular languages. The simplification operation consists in finding E-equivalent terms recognized in $\mathcal{A}$ by different states and then by merging those states together. The merging of states is performed using renaming of a state in a tree automaton.

Definition 6 (Renaming of a state in a tree automaton). *Let $\mathcal{Q}, \mathcal{Q}'$ be set of states, $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, and α a function $\alpha : \mathcal{Q} \mapsto \mathcal{Q}'$. We denote by $\mathcal{A}\alpha$ the tree automaton where every occurrence of q is replaced by $\alpha(q)$ in $\mathcal{Q}$, $\mathcal{Q}_f$ and in every left and right-hand side of every transition of Δ.*

If there exists a bijection α such that $\mathcal{A} = \mathcal{A}'\alpha$ then $\mathcal{A}$ and $\mathcal{A}'$ are said to be *equivalent modulo renaming*. Now we define the *simplification relation* which merges states in a tree automaton according to an equation. Note that it is not required for equations of E to be linear.

Definition 7 (Simplification relation). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton and E be a set of equations. For $s = t \in E$, $\sigma \in \Sigma(\mathcal{Q}, \mathcal{X})$, $q_a, q_b \in \mathcal{Q}$ such that $s\sigma \to_\mathcal{A}^{\not\epsilon *} q_a$, $t\sigma \to_\mathcal{A}^{\not\epsilon *} q_b$, and $q_a \neq q_b$ then $\mathcal{A}$ can be* simplified *into $\mathcal{A}' = \mathcal{A}\{q_b \mapsto q_a\}$, denoted by $\mathcal{A} \leadsto_E \mathcal{A}'$.*

Example 3. Let $E = \{s(s(x)) = s(x)\}$ and $\mathcal{A}$ be the tree automaton with set of transitions $\Delta = \{a \to q_0, s(q_0) \to q_1, s(q_1) \to q_2\}$. We can perform a simplification step using the equation $s(s(x)) = s(x)$ because we found a substitution $\sigma = \{x \mapsto q_0\}$ such that: $s(s(x))\sigma \to_\mathcal{A}^{\not\epsilon *} q_2$ and $s(x)\sigma \to_\mathcal{A}^{\not\epsilon *} q_1$ Hence, $\mathcal{A} \leadsto_E \mathcal{A}' = \mathcal{A}\{q_2 \mapsto q_1\}$[2]

[2] or $\{q_1 \mapsto q_2\}$, any of q_1 or q_2 can be used for renaming.

As stated in [13], simplification $\leadsto_E$ is a terminating relation (each step suppresses a state) and it enlarges the language recognized by a tree automaton, *i.e.* if $\mathcal{A} \leadsto_E \mathcal{A}'$ then $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$. Furthermore, no matter how simplification steps are performed, the obtained automata are equivalent modulo state renaming. In the following, $\mathcal{A} \leadsto_E^! \mathcal{A}'$ denotes that $\mathcal{A} \leadsto_E^* \mathcal{A}'$ and $\mathcal{A}'$ is irreducible by $\leadsto_E$. We denote by $\mathcal{S}_E(\mathcal{A})$ any automaton $\mathcal{A}'$ such that $\mathcal{A} \leadsto_E^! \mathcal{A}'$.

Theorem 1 (Simplified Tree Automata [13]). *Let $\mathcal{A}, \mathcal{A}'_1, \mathcal{A}'_2$ be tree automata and E be a set of equations. If $\mathcal{A} \leadsto_E^! \mathcal{A}'_1$ and $\mathcal{A} \leadsto_E^! \mathcal{A}'_2$ then $\mathcal{A}'_1$ and $\mathcal{A}'_2$ are equivalent modulo state renaming.*

4.4 The full Completion Algorithm

Definition 8 (Automaton completion). *Let $\mathcal{A}$ be a tree automaton, $\mathcal{R}$ a left-linear TRS and E a set of equations.*

- $\mathcal{A}^0_{\mathcal{R},E} = \mathcal{A}$
- $\mathcal{A}^{n+1}_{\mathcal{R},E} = \mathcal{S}_E\left(\mathcal{C}_\mathcal{R}(\mathcal{A}^n_{\mathcal{R},E})\right)$, *for* $n \geq 0$

If there exists $k \in \mathbb{N}$ such that $\mathcal{A}^k_{\mathcal{R},E} = \mathcal{A}^{k+1}_{\mathcal{R},E}$, then we denote $\mathcal{A}^k_{\mathcal{R},E}$ by $\mathcal{A}^*_{\mathcal{R},E}$.

In practice, checking if $CP(\mathcal{R}, \mathcal{A}^k_{\mathcal{R},E}) = \emptyset$ is sufficient to know that $\mathcal{A}^k_{\mathcal{R},E}$ is a fixpoint. However, a fixpoint cannot always be finitely reached[3]. To ensure termination, one can provide a set of approximating equations to overcome infinite rewriting and completion divergence.

Example 4. Let $\mathcal{R} = \{f(x,y) \to f(s(x), s(y))\}$, $E = \{s(s(x)) = s(x)\}$ and $\mathcal{A}^0$ be the tree automaton with set of transitions $\Delta = \{f(q_a, q_b) \to q_0), a \to q_a, b \to q_b\}$, *i.e.* $\mathcal{L}(\mathcal{A}^0) = \{f(a,b)\}$. The completion ends after two completion steps on $\mathcal{A}^2_{\mathcal{R},E}$ which is a fixpoint. Completion steps are summed up in the following table. To simplify the presentation, we do not repeat the common transitions: $\mathcal{A}^i_{\mathcal{R},E}$ and $\mathcal{C}_\mathcal{R}(\mathcal{A}^i)$ columns are supposed to contain all transitions of $\mathcal{A}^0, \ldots, \mathcal{A}^{i-1}_{\mathcal{R},E}$. The automaton $\mathcal{A}^1_{\mathcal{R},E}$ is exactly $\mathcal{C}_\mathcal{R}(\mathcal{A}^0)$ since simplification by equations do not apply. Simplification has been applied on $\mathcal{C}_\mathcal{R}(\mathcal{A}^1_{\mathcal{R},E})$ to obtain $\mathcal{A}^2_{\mathcal{R},E}$.

$\mathcal{A}^0$	$\mathcal{C}_\mathcal{R}(\mathcal{A}^0)$	$\mathcal{A}^1_{\mathcal{R},E}$	$\mathcal{C}_\mathcal{R}(\mathcal{A}^1_{\mathcal{R},E})$	$\mathcal{A}^2_{\mathcal{R},E}$
$f(q_a, q_b) \to q_0$	$f(q_1, q_2) \to q_3$	$f(q_1, q_2) \to q_3$	$f(q_4, q_5) \to q_6$	$f(q_1, q_2) \to q_6$
$a \to q_a$	$s(q_a) \to q_1$	$s(q_a) \to q_1$	$s(q_1) \to q_4$	$s(q_1) \to q_1$
$b \to q_b$	$s(q_b) \to q_2$	$s(q_b) \to q_2$	$s(q_2) \to q_5$	$s(q_2) \to q_2$
	$q_3 \to q_0$	$q_3 \to q_0$	$q_6 \to q_3$	

Now, we recall the lower and upper bound theorems. Tree automata completion of automaton $\mathcal{A}$ with TRS $\mathcal{R}$ and set of equations E is lower bounded by $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ and upper bounded by $\mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$. The lower bound theorem ensures that the completed automaton $\mathcal{A}^*_{\mathcal{R},E}$ recognizes all $\mathcal{R}$-reachable terms (but not all $\mathcal{R}/E$-reachable terms). The upper bound theorem guarantees that all terms recognized by $\mathcal{A}^*_{\mathcal{R},E}$ are only $\mathcal{R}/E$-reachable terms.

[3] See [10], for classes of $\mathcal{R}$ for which a fixpoint always exists.

Theorem 2 (Lower bound [13]). *Let $\mathcal{R}$ be a left-linear TRS, $\mathcal{A}$ be a tree automaton and E be a set of equations. If completion terminates on $\mathcal{A}^*_{\mathcal{R},E}$ then $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$.*

The upper bound theorem states the precision result of completion. It is defined using the $\mathcal{R}/E$-coherence property. The intuition behind $\mathcal{R}/E$-coherence is the following: in the tree automaton ϵ-transitions represent rewriting steps and normalized transitions recognize E-equivalence classes. More precisely, in a $\mathcal{R}/E$-coherent tree automaton, if two terms s, t are recognized into the same state q using only normalized transitions then they belong to the same E-equivalence class. Otherwise, if at least one ϵ-transition is necessary to recognize, say, t into q then at least one step of rewriting was necessary to obtain t from s.

Theorem 3 (Upper bound [13]). *Let $\mathcal{R}$ be a left-linear TRS, E a set of equations and $\mathcal{A}$ a $\mathcal{R}/E$-coherent tree automaton. For any $i \in \mathbb{N}$: $\mathcal{L}(\mathcal{A}^i_{\mathcal{R},E}) \subseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$ and $\mathcal{A}^i_{\mathcal{R},E}$ is $\mathcal{R}/E$-coherent.*

5 Termination criterion for a given set of equations

Given a set of equations E, the effect of the simplification with E on a tree automaton is to merge two distinct states recognizing instances of the left and right-hand side for all the equations of E. In this section, we give a sufficient condition on E and on the completed tree automata $\mathcal{A}^i_{\mathcal{R},E}$ for the tree automata completion to always terminate. The intuition behind this condition is simple: if the set of equivalence classes for E, *i.e.* $\mathcal{T}(\mathcal{F})/_{=_E}$, is finite then so should be the set of new states used in completion. However, this is not true in general because simplification of an automaton with E does not necessarily merge all E-equivalent terms.

Example 5. Let $\mathcal{A}$ be the tree automaton with set of transitions $a \to q$, $\mathcal{R} = \{a \to c\}$ and let $E = \{a = b, b = c\}$. The set of transitions of $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$ is $\{a \to q, c \to q', q' \to q\}$. We have $a =_E c$, $a \in \mathcal{L}^{\not\epsilon}(\mathcal{C}_{\mathcal{R}}(\mathcal{A}), q)$ and $c \in \mathcal{L}^{\not\epsilon}(\mathcal{C}_{\mathcal{R}}(\mathcal{A}), q')$ but on the automaton $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$, no simplification situation (as described by Definition 7), can be found because the term b is not recognized by $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$. Hence, the simplified automaton is $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$ where a and c are recognized by different states.

There is no simple solution to have a simplification algorithm merging all states recognizing E-equivalent terms (see Section 6). Having a complete automaton $\mathcal{A}$ solve the above problem but leads to rough approximations (see [11]). In the next section, we propose to give some simple restrictions on E to ensure that completion terminates. In Section 5.2, we will see how those restrictions can easily be met for "functional" TRS, *i.e.* a typed first-order functional program translated into a TRS.

5.1 General criterion

What Example 5 shows is that, for a simplification with E to apply, it is necessary that both sides of the equation are recognized by the tree automaton. In the following, we will define a set E^c of *contracting* equations so that this property is true. What Example 5 does not show is that, by default, tree automata are not E-compatible. In particular, any non $\not\epsilon$-deterministic automaton does not satisfy the reflexivity of $=_E$. For instance, if an automaton $\mathcal{A}$ has two transitions $a \to q_1$ and $a \to q_2$, since $a =_E a$ for all E, for $\mathcal{A}$ to be E-compatible we should have $q_1 = q_2$. To enforce $\not\epsilon$-determinism by automata simplification, we define a set of *reflexivity equations* as follows.

Definition 9 (Set of reflexivity equations E^r). *For a given set of symbols $\mathcal{F}$, $E^r = \{f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \mid f \in \mathcal{F}$, and arity of f is $n\}$, where $x_1 \ldots x_n$ are pairwise distinct variables.*

Note that for all set of equations E, the relation $=_E$ is trivially equivalent to $=_{E \cup E^r}$. Furthermore, simplification with E^r transforms all automaton into an $\not\epsilon$-deterministic automaton, as stated in the following lemma.

Lemma 1. *For all tree automaton $\mathcal{A}$ and all set of equation E, if $E \supseteq E^r$ and $\mathcal{A} \rightsquigarrow_E^! \mathcal{A}'$ then $\mathcal{A}'$ is $\not\epsilon$-deterministic.*

Proof. Shown by induction on the height of terms (see [11] for details). □

We now define sets of contracting equations. Such sets are defined for a set of symbols $\mathcal{K}$ which can be a subset of $\mathcal{F}$. This will be used later to restrict contracting equations to the subset of constructor symbols of $\mathcal{F}$.

Definition 10 (Sets of contracting equations for $\mathcal{K}$, $E^c_{\mathcal{K}}$). *Let $\mathcal{K} \subseteq \mathcal{F}$. A set of equations is contracting for $\mathcal{K}$, denoted by $E^c_{\mathcal{K}}$, if all equations of $E^c_{\mathcal{K}}$ are of the form $u = u|_p$ with $u \in \mathcal{T}(\mathcal{K}, \mathcal{X})$ a linear term, $p \neq \lambda$, and if the set of normal forms of $\mathcal{T}(\mathcal{K})$ w.r.t. the TRS $\overrightarrow{E^c_{\mathcal{K}}} = \{u \to u|_p \mid u = u|_p \in E^c_{\mathcal{K}}\}$ is finite.*

Contracting equations, if defined on $\mathcal{F}$, define an upper bound on the number of states of a simplified automaton.

Lemma 2. *Let $\mathcal{A}$ be a tree automaton and $E^c_{\mathcal{F}}$ a set of contracting equations for $\mathcal{F}$. If $E \supseteq E^c_{\mathcal{F}} \cup E^r$ then the simplified automaton $\mathcal{S}_E(\mathcal{A})$ is an $\not\epsilon$-deterministic automaton having no more states than terms in $\textsc{Irr}(\overrightarrow{E^c_{\mathcal{F}}})$.*

Proof. First, assume for all state q of $\mathcal{S}_E(\mathcal{A})$, $\mathcal{L}^{\not\epsilon}(\mathcal{S}_E(\mathcal{A}), q) \cap \textsc{Irr}(\overrightarrow{E^c_{\mathcal{F}}}) = \emptyset$. Then, for all terms s such that $s \to^{\not\epsilon *}_{\mathcal{S}_E(\mathcal{A})} q$, we know that s is not in normal form w.r.t. $\overrightarrow{E^c_{\mathcal{F}}}$. As a result, the left-hand side of an equation of $E^c_{\mathcal{F}}$ can be applied to s. This means that there exists an equation $u = u|_p$, a ground context C and a substitution θ such that $s = C[u\theta]$. Furthermore, since $s \to^{\not\epsilon *}_{\mathcal{S}_E(\mathcal{A})} q$, we know that $C[u\theta] \to^{\not\epsilon *}_{\mathcal{S}_E(\mathcal{A})} q$ and that there exists a state q' such that $C[q'] \to^{\not\epsilon *}_{\mathcal{S}_E(\mathcal{A})} q$

and $u\theta \to^{\not\epsilon *}_{\mathcal{S}_E(\mathcal{A})} q'$. From $u\theta \to^{\not\epsilon *}_{\mathcal{S}_E(\mathcal{A})} q'$, we know that all subterms of $u\theta$ are recognized by at least one state in $\mathcal{S}_E(\mathcal{A})$. Thus, there exists a state q'' such that $u|_p\theta \to^{\not\epsilon *}_{\mathcal{S}_E(\mathcal{A})} q''$. We thus have a situation of application of the equation $u = u|_p$ in the automaton. Since $\mathcal{S}_E(\mathcal{A})$ is simplified, we thus know that $q' = q''$. As mentioned above, we know that $C[q'] \to^{\not\epsilon *}_{\mathcal{S}_E(\mathcal{A})} q$. Hence $C[u|_p\theta] \to^{\not\epsilon *}_{\mathcal{S}_E(\mathcal{A})} C[q'] \to^{\not\epsilon *}_{\mathcal{S}_E(\mathcal{A})} q$. If $C[u|_p\theta]$ is not in normal form w.r.t. $\overrightarrow{E^c_{\mathcal{F}}}$ then we can do the same reasoning on $C[u|_p\theta] \to^{\not\epsilon *}_{\mathcal{S}_E(\mathcal{A})} q$ until getting a term that is in normal form w.r.t. $\overrightarrow{E^c_{\mathcal{F}}}$ and recognized by the same state q. Thus, this contradicts the fact that $\mathcal{S}_E(\mathcal{A})$ recognizes no term of $\textsc{Irr}(\overrightarrow{E^c_{\mathcal{F}}})$.

Then, by definition of $E^c_{\mathcal{F}}$, $\textsc{Irr}(\overrightarrow{E^c_{\mathcal{F}}})$ is finite. Let $\{t_1, \ldots, t_n\}$ be the subset of $\textsc{Irr}(\overrightarrow{E^c_{\mathcal{F}}})$ recognized by $\mathcal{S}_E(\mathcal{A})$. Let $q_1, \ldots, q_n$ be the states recognizing $t_1, \ldots, t_n$ respectively. We know that there is a finite set of states recognizing $t_1, \ldots, t_n$ because $E \supseteq E^r$ and Lemma 1 entails that $\mathcal{S}_E(\mathcal{A})$ is $\not\epsilon$deterministic. Now, for all terms s recognized by a state q in $\mathcal{S}_E(\mathcal{A})$, *i.e.* $s \to^{\not\epsilon *}_{\mathcal{S}_E(\mathcal{A})} q$, we can use a reasoning similar to the one carried out above and show that q is equal to one state of $\{q_1, \ldots, q_n\}$ recognizing normal forms of $\overrightarrow{E^c_{\mathcal{F}}}$ in $\mathcal{S}_E(\mathcal{A})$. Finally, there are at most $card(\textsc{Irr}(\overrightarrow{E^c_{\mathcal{F}}}))$ states in $\mathcal{S}_E(\mathcal{A})$. □

Now it is possible to state the Theorem guaranteeing the termination of completion if the set of equations E contains a set of contracting equations $E^c_{\mathcal{F}}$ for $\mathcal{F}$ and a set of reflexivity equations.

Theorem 4. *Let $\mathcal{A}$ be a tree automaton, $\mathcal{R}$ a left linear TRS and E a set of equations. If $E \supseteq E^r \cup E^c_{\mathcal{F}}$, then completion of $\mathcal{A}$ by $\mathcal{R}$ and E terminates.*

Proof. For completion to diverge it must produce infinitely many new states. This is impossible if E contains $E^c_{\mathcal{F}}$ and E^r (see Lemma 2). □

5.2 Criterion for Functional TRSs

Now, we consider functional programs viewed as TRSs. We assume that such TRSs are left-linear, which is a common assumption on TRSs obtained from functional programs [2]. In this section, we will restrict ourselves to sufficiently complete TRSs obtained from functional programs and will refer to them as *functional TRSs*. For TRSs representing functional programs, defining contracting equations of $E^c_{\mathcal{C}}$ on $\mathcal{C}$ rather than on $\mathcal{F}$ is enough to guarantee termination of completion. This is more convenient and also closer to what is usually done in static analysis where abstractions are usually defined on data and not on function applications. Since the TRSs we consider are sufficiently complete, any term of $\mathcal{T}(\mathcal{F})$ can be rewritten into a data-term of $\mathcal{T}(\mathcal{C})$. As above, using equations of $E^c_{\mathcal{C}}$ we are going to ensure that the data-terms of the computed languages will be recognized by a bounded set of states. To lift-up this property to $\mathcal{T}(\mathcal{F})$ it is enough to ensure that $\forall s, t \in \mathcal{T}(\mathcal{F})$ if $s \to_R t$ then s and t are recognized by equivalent states. This is the role of the set of equations E_R.

Definition 11 ($E_{\mathcal{R}}$). *Let $\mathcal{R}$ be a TRS, the set of $\mathcal{R}$-equations is $E_{\mathcal{R}} = \{l = r \mid l \to r \in \mathcal{R}\}$.*

Theorem 5. *Let $\mathcal{A}_0$ be a tree automaton, $\mathcal{R}$ a sufficiently complete left-linear TRS and E a set of equations. If $E \supseteq E^r \cup E^c_{\mathcal{C}} \cup E_{\mathcal{R}}$ with $E^c_{\mathcal{C}}$ contracting then completion of $\mathcal{A}_0$ by $\mathcal{R}$ and E terminates.*

Proof. Firstly, to show that the number of states recognizing terms of $\mathcal{T}(\mathcal{C})$ is finite we can do a proof similar to the one of Lemma 2 . Let $G \subseteq \mathcal{T}(\mathcal{C})$ be the finite set of normal forms of $\mathcal{T}(\mathcal{C})$ w.r.t. $\overrightarrow{E^c_{\mathcal{C}}}$. Since $E \supseteq E^r \cup E^c_{\mathcal{C}}$, like in the proof of Lemma 2, we can show that in any completed automaton, terms of $\mathcal{T}(\mathcal{C})$ are recognized by no more states than terms in G. Secondly, since $\mathcal{R}$ is sufficiently complete, for all terms $s \in \mathcal{T}(\mathcal{F}) \setminus \mathcal{T}(\mathcal{C})$ we know that there exists a term $t \in \mathcal{T}(\mathcal{C})$ such that $s \to_{\mathcal{R}}{}^* t$. The fact that $E \supseteq E_{\mathcal{R}}$ guarantees that s and t will be recognized by equivalent states in the completed (and simplified) automaton. Since the number of states necessary to recognize $\mathcal{T}(\mathcal{C})$ is finite, so is the number of states necessary to recognize terms of $\mathcal{T}(\mathcal{F})$. □

Finally, to exploit the types of the functional program, we now see $\mathcal{F}$ as a many-sorted signature whose set of sorts is $\mathcal{S}$. Each symbol $f \in \mathcal{F}$ is associated to a profile $f : S_1 \times \ldots \times S_k \mapsto S$ where $S_1, \ldots, S_k, S \in \mathcal{S}$ and k is the arity of f. Well-sorted terms are inductively defined as follows: $f(t_1, \ldots, t_k)$ is a well-sorted term of sort S if $f : S_1 \times \ldots \times S_k \mapsto S$ and $t_1, \ldots, t_k$ are well-sorted terms of sorts $S_1, \ldots, S_k$, respectively. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{X})^{\mathcal{S}}$, $\mathcal{T}(\mathcal{F})^{\mathcal{S}}$ and $\mathcal{T}(\mathcal{C})^{\mathcal{S}}$ the set of well-sorted terms, ground terms and constructor terms, respectively. Note that we have $\mathcal{T}(\mathcal{F}, \mathcal{X})^{\mathcal{S}} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$, $\mathcal{T}(\mathcal{F})^{\mathcal{S}} \subseteq \mathcal{T}(\mathcal{F})$ and $\mathcal{T}(\mathcal{C})^{\mathcal{S}} \subseteq \mathcal{T}(\mathcal{C})$. We assume that $\mathcal{R}$ and E are *sort preserving*, *i.e.* that for all rule $l \to r \in R$ and all equation $u = v \in E$, $l, r, u, v \in \mathcal{T}(\mathcal{F}, \mathcal{X})^{\mathcal{S}}$, l and r have the same sort and so do u and v. Note that well-typedness of the functional program entails the well-sortedness of $\mathcal{R}$. We still assume that the (sorted) TRS is sufficiently complete, which is defined in a similar way except that it holds only for well-sorted terms, *i.e.* for all $s \in \mathcal{T}(\mathcal{F})^{\mathcal{S}}$ there exists a term $t \in \mathcal{T}(\mathcal{C})^{\mathcal{S}}$ such that $s \to_{\mathcal{R}}{}^* t$. We slightly refine the definition of contracting equations as follows. For all sort S, if S has a unique constant symbol we note it c^S.

Definition 12 (Set $E^c_{\mathcal{K},\mathcal{S}}$ of contracting equations for $\mathcal{K}$ and $\mathcal{S}$). *Let $\mathcal{K} \subseteq \mathcal{F}$. The set of well-sorted equations $E^c_{\mathcal{K},\mathcal{S}}$ is* contracting *(for $\mathcal{K}$) if its equations are of the form (a) $u = u|_p$ with u linear and $p \neq \Lambda$, or (b) $u = c^S$ with u of sort S, and if the set of normal forms of $\mathcal{T}(\mathcal{K})^{\mathcal{S}}$ w.r.t. the TRS $\overrightarrow{E^c_{\mathcal{K},\mathcal{S}}} = \{u \to v \mid u = v \in E^c_{\mathcal{K},\mathcal{S}} \wedge (v = u|_p \vee v = c^S)\}$ is finite.*

The termination theorem for completion of sorted TRSs is similar to the previous one except that it needs $\mathcal{R}/E$-coherence of $\mathcal{A}_0$ to ensure that terms recognized by completed automata are well-sorted (see [11] for proof).

Theorem 6. *Let $\mathcal{A}_0$ be a tree automaton recognizing well-sorted terms, $\mathcal{R}$ a sufficiently complete sort-preserving left-linear TRS and E a sort-preserving set*

of equations. If $E \supseteq E^r \cup E^c_{\mathcal{C},\mathcal{S}} \cup E_{\mathcal{R}}$ *with* $E^c_{\mathcal{C},\mathcal{S}}$ *contracting and* $\mathcal{A}_0$ *is* $\mathcal{R}/E$*-coherent then completion of* $\mathcal{A}_0$ *by* $\mathcal{R}$ *and* E *terminates.*

5.3 Experiments

The objective of data-flow analysis is to predict the set of all program states reachable from a language of initial function calls, *i.e.* to over-approximate $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ where $\mathcal{R}$ represents the functional program and $\mathcal{A}$ the language of initial function calls. In this setting, we automatically compute an automaton $\mathcal{A}^*_{\mathcal{R},E}$ over-approximating $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. But we can do more. Since we are dealing with left-linear TRS, it is possible to build $\mathcal{A}_{\text{IRR}(\mathcal{R})}$ recognizing $\text{IRR}(\mathcal{R})$. Finally, since tree automata are closed under all boolean operations, we can compute an approximation of all the results of the function calls by computing the tree automaton recognizing the intersection between $\mathcal{A}^*_{\mathcal{R},E}$ and $\mathcal{A}_{\text{IRR}(\mathcal{R})}$.

Here is an example of application of those theorems. Completions are performed using Timbuk. All the $\mathcal{A}_{\text{IRR}(\mathcal{R})}$ automata and intersections were performed using Taml. Details can be found in [14].

```
Ops append:2 rev:1 nil:0 cons:2 a:0 b:0   Vars X Y Z U Xs
TRS R
append(nil,X)->X      append(cons(X,Y),Z)->cons(X,append(Y,Z))
rev(nil)->nil         rev(cons(X,Y))->append(rev(Y),cons(X,nil))

Automaton A0 States q0 qla qlb qnil qf qa qb Final States q0 Transitions
rev(qla)->q0          cons(qb,qnil)->qlb    cons(qa,qla)->qla    nil->qnil
cons(qa,qlb)->qla     a->qa                 cons(qb,qlb)->qlb    b->qb

Equations E Rules     cons(X,cons(Y,Z))=cons(Y,Z)  %%% Ec
%%% E_R                                         %%% E^r
append(nil,X)=X                                 rev(X)=rev(X)
append(cons(X,Y),Z)=cons(X,append(Y,Z))         cons(X,Y)=cons(X,Y)
rev(nil)=nil                                    append(X,Y)=append(X,Y)
rev(cons(X,Y))=append(rev(Y),cons(X,nil))       a=a  b=b  nil=nil
```

In this example, the TRS $\mathcal{R}$ encodes the classical *reverse* and *append* functions. The language recognized by automaton $\mathcal{A}_0$ is the set of terms of the form $rev([a, a, \ldots, b, b, \ldots])$. Note that there are at least one a and one b in the list. We assume that $\mathcal{S} = \{T, list\}$ and sorts for symbols are the following: $a : T$, $b : T$, $nil : list$, $cons : T \times list \mapsto list$, $append : list \times list \mapsto list$ and $rev : list \mapsto list$. Now, to use Theorem 6, we need to prove each of its assumptions. The set E of equations contains $E_{\mathcal{R}}$, E^r and $E^c_{\mathcal{C},\mathcal{S}}$. The set of Equations $E^c_{\mathcal{C},\mathcal{S}}$ is contracting because the automaton $\mathcal{A}_{\text{IRR}(\overrightarrow{E^c_{\mathcal{C},\mathcal{S}}})}$ recognizes a finite language. This automaton can be computed using Taml: it is the intersection between the automaton $\mathcal{A}_{\mathcal{T}(\mathcal{C})^{\mathcal{S}}}$[4] recognising $\mathcal{T}(\mathcal{C})^{\mathcal{S}}$ and the automaton $\mathcal{A}_{\text{IRR}(\{cons(X,cons(Y,Z))\to cons(Y,Z)\})}$:

[4] Such an automaton has one state per sort and one transition per constructor. For instance, on our example $\mathcal{A}_{\mathcal{T}(\mathcal{C})^{\mathcal{S}}}$ will have transitions: $a \to qT$, $b \to qT$, $cons(qT, qlist) \to qlist$ and $nil \to qlist$.

States `q2 q1 q0` **Final States** `q0 q1 q2`
Transitions `b->q2 a->q2 nil->q1 cons(q2,q1)->q0`

The language of $\mathcal{A}_0$ is well-sorted and E and $\mathcal{R}$ are sort preserving. We can prove sufficient completeness of $\mathcal{R}$ on $\mathcal{T}(\mathcal{F})^{\mathcal{S}}$ using, for instance, Maude [6] or even Timbuk [9] itself. The last assumption of Theorem 6 to prove is that $\mathcal{A}_0$ is $\mathcal{R}/E$-coherent. This can be shown by remarking that each state q of $\mathcal{A}_0$ recognizes at least one term and if $s \to^{\not\varepsilon *}_{\mathcal{A}_0} q$ and $t \to^{\not\varepsilon *}_{\mathcal{A}_0} q$ then $s =_E t$. For instance $cons(b, cons(b, nil)) \to^{\not\varepsilon *}_{\mathcal{A}_0} q_{lb}$ and $cons(b, nil) \to^{\not\varepsilon *}_{\mathcal{A}_0} q_{lb}$ and $cons(b, cons(b, nil)) =_E cons(b, nil)$. Thus, completion is guaranteed to terminate: after 4 completion steps (7 ms) we obtain a fixpoint automaton $\mathcal{A}^*_{\mathcal{R},E}$ with 11 transitions. To restrain the language to normal forms it is enough to compute the intersection with $\text{IRR}(R)$. Since we are dealing with sufficiently complete TRSs, we know that $\text{IRR}(R) \subseteq \mathcal{T}(\mathcal{C})^{\mathcal{S}}$. Thus, we can use again $\mathcal{A}_{\mathcal{T}(\mathcal{C})^s}$ for the intersection that is:

States `q3 q2 q1 q0` **Final States** `q3` **Transitions** `a->q0 nil->q1 b->q2 cons(q0,q1)->q3 cons(q0,q3)->q3 cons(q2,q1)->q3 cons(q2,q3)->q3`

which recognizes any (non empty) flat list of a and b. Thus, our analysis preserved the property that the result cannot be the empty list but lost the order of the elements in the list. This is not surprising because the equation `cons(X, cons(Y, Z))=cons(X, Z)` makes $cons(a, cons(b, nil))$ equal to $cons(a, nil)$. It is possible to refine by hand $E^c_{\mathcal{C},\mathcal{S}}$ using the following equations: `cons(a,cons(a,X))=cons(a,X)`, `cons(b,cons(b,X))=cons(b,X)`, `cons(a,cons(b,cons(a,X)))=cons(a,X)`. This set of equations avoids the previous problem. Again, E verifies the conditions of Theorem 6 and completion is still guaranteed to terminate. The result is the automaton $\mathcal{A}'^*_{\mathcal{R},E}$ having 19 transitions. This time, intersection with $\mathcal{A}_{\mathcal{T}(\mathcal{C})^s}$ gives:

States `q4 q3 q2 q1 q0` **Final States** `q4` **Transitions** `a->q1 b->q3 nil->q0 cons(q1,q0)->q2 cons(q1,q2)->q2 cons(q3,q2)->q4 cons(q3,q4)->q4`

This automaton exactly recognizes lists of the form $[b, b, \ldots, a, a, \ldots]$ with at least one b and one a, as expected. Hopefully, refinement of equations can be automatized in completion [3] and can be used here, see [14] for examples. More examples can be found in the Timbuk 3.1 source distribution.

6 Conclusion and further research

In this paper we defined a criterion on the set of approximation equations to guarantee termination of the tree automata completion. When dealing with, so called, functional TRS this criterion is close to what is generally expected in static analysis and abstract interpretation: a finite model for an infinite set of data-terms. This work is a first step to use completion for static analysis of functional programs. There remains some interesting points to address.

Dealing with higher-order functions. Higher-order functions can be encoded into first order TRS using a simple encoding borrowed from [17]: defined symbols become constants, constructor symbols remain the same, and an additional *application* operator '@' of arity 2 is introduced. On all the examples of [21], completion and this simple encoding produces exactly the same results [14].

Dealing with evaluation strategies. The technique proposed here, as well as [21], over-approximates the set of results for all evaluation strategies. As far as we know, no static analysis technique for functional programs can take into account evaluation strategies. However, it is possible to restrict the completion algorithm to recognize only innermost descendants [14], *i.e.* call-by-value results. If the approximation is precise enough, any non terminating program with call-by-value will have an empty set of results. An open research direction is to use this to prove non termination of functional programs by call-by-value strategy.

Dealing with built-in types. Values manipulated by *real* functional programs are not always terms or trees. They can be numerals or be terms embedding numerals. In [12], it has been shown that completion can compute over-approximations of reachable terms embedding built-in terms. The structural part of the term is approximated using tree automata and the built-in part is approximated using lattices and abstract interpretation.

Besides, there remain some interesting theoretical points to solve. In section 5, we saw that having a finite $\mathcal{T}(\mathcal{F})/_{=_E}$ is not enough to guarantee the termination of completion. This is due to the fact that the simplification algorithm does not merge all states recognizing E-equivalent terms. Having a simplification algorithm ensuring this property is not trivial. First, the theory defined by E has to be decidable. Second, even if E is decidable, finding all the E-equivalent terms recognized by the tree automaton is an open problem. Furthermore, proving that $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite, is itself difficult. This question is undecidable in general [23], but can be answered for some particular E. For instance, if E can be oriented into a TRS $\mathcal{R}$ which is terminating, confluent and such that $\textsc{Irr}(\mathcal{R})$ is finite then $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite [23].

Acknowledgments Many thanks to the referees for their detailed comments.

References

1. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santos Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the automated validation of internet security protocols and applications. In *CAV'2005*, volume 3576 of *LNCS*, pages 281–285. Springer, 2005.
2. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
3. Y. Boichut, B. Boyer, T. Genet, and A. Legay. Equational Abstraction Refinement for Certified Tree Regular Model Checking. In *ICFEM'12*, volume 7635 of *LNCS*. Springer, 2012.

4. Y. Boichut, R. Courbis, P.-C. Héam, and O. Kouchnarenko. Handling non left-linear rules when completing tree automata. *IJFCS*, 20(5), 2009.
5. Y. Boichut, T. Genet, T. Jensen, and L. Leroux. Rewriting Approximations for Fast Prototyping of Static Analyzers. In *RTA*, volume 4533 of *LNCS*, pages 48–62. Springer, 2007.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
7. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://tata.gforge.inria.fr`, 2008.
8. G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasonning*, 33 (3-4):341–383, 2004.
9. T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proc. 9th RTA Conf., Tsukuba (Japan)*, volume 1379 of *LNCS*, pages 151–165. Springer-Verlag, 1998.
10. T. Genet. Reachability analysis of rewriting for software verification. Université de Rennes 1, 2009. Habilitation document, `http://www.irisa.fr/celtique/genet/publications.html`.
11. T. Genet. Towards Static Analysis of Functional Programs using Tree Automata Completion. Technical report, INRIA, 2013. `http://hal.archives-ouvertes.fr/hal-00921814/PDF/main.pdf`.
12. T. Genet, T. Le Gall, A. Legay, and V. Murat. A Completion Algorithm for Lattice Tree Automata. In *CIAA'13*, volume 7982 of *LNCS*, pages 134–145, 2013.
13. T. Genet and R. Rusu. Equational tree automata completion. *Journal of Symbolic Computation*, 45:574–597, 2010.
14. T. Genet and Y. Salmon. Tree Automata Completion for Static Analysis of Functional Programs. Technical report, INRIA, 2013. `http://hal.archives-ouvertes.fr/hal-00780124/PDF/main.pdf`.
15. A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. On tree automata that certify termination of left-linear term rewriting systems. In *RTA'05*, volume 3467 of *LNCS*, pages 353–367. Springer, 2005.
16. F. Jacquemard. Decidable approximations of term rewriting systems. In H. Ganzinger, editor, *Proc. 7th RTA Conf., New Brunswick (New Jersey, USA)*, pages 362–376. Springer-Verlag, 1996.
17. N. D. Jones. Flow analysis of lazy higher-order functional programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 103–122. Ellis Horwood, Chichester, England, 1987.
18. Naoki Kobayashi. Model checking higher-order programs. *J. ACM*, 60(3):20, 2013.
19. A. Lisitsa. Finite Models vs Tree Automata in Safety Verification. In *RTA'12*, volume 15 of *LIPIcs*, pages 225–239, 2012.
20. F. Oehl, G. Cécé, O. Kouchnarenko, and D. Sinclair. Automatic Approximation for the Verification of Cryptographic Protocols. In *Proc. of FASE'03*, volume 2629 of *LNCS*, pages 34–48. Springer-Verlag, 2003.
21. L. Ong and S. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL'11*, 2011.
22. T. Takai. A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In *Proc. 15th RTA Conf., Aachen (Germany)*, volume 3091 of *LNCS*, pages 119–133. Springer, 2004.
23. S. Tison. Finiteness of the set of E-equivalence classes is undecidable, 2010. Private communication.

Key-Secrecy of PACE with OTS/CafeOBJ

Dominik Klein

Bundesamt für Sicherheit in der Informationstechnik (BSI)
Godesberger Allee 185–189
53175 Bonn, Germany
Dominik.Klein@bsi.bund.de

Abstract. The ICAO-standardized Password Authenticated Connection Establishment (PACE) protocol is used all over the world to secure electronic passports. We verify key-secrecy of PACE by modelling it as an Observational Transition System (OTS) in CafeOBJ.

1 Introduction

Cryptographic primitives such as encryption mechanisms, hash functions or message authentication codes undergo the scrutiny of a large community of researchers. While their mathematical foundations might not yet be understood in full detail, sudden ground-breaking attacks on them have been few. Using these primitives as building blocks to construct security protocols however is another, difficult challenge. In fact, it was often that erroneous protocol specifications and design decisions lead to attacks, despite the usage of well-known cryptographic primitives suggesting a high level of security. Often cited for a subtle, easily overlooked error is [12], but [6] contains an impressive list of failed attempts to design secure protocols. Formally proving properties of a protocol to exclude subtle attacks is one important step — amongst others — in the construction of security protocols.

Password Authenticated Connection Establishment (PACE) [3] is used all over the world to secure communication with electronic passports. Here, PACE replaces the older Basic Access Control (BAC) where security concerns with low-entropy passwords occurred. RFID has been used in electronic passports over contact-based solutions for the ability to keep passport formats and for reasons of durability. This raises concerns of citizens that passports enable secret tracking or that criminals secretly read out sensitive biometric information. Also PACE is used in national id-cards that enable secure two-factor authentication for e-commerce. PACE protects the highly valuable biometric information of the document holder. For all these reasons, ensuring trust in PACE is of uttermost importance: For once to secure the communication, but also to increase acceptance of citizens for electronic passports.

Our contribution is threefold. First, we show key secrecy of PACE itself, facilitating trust in the protocol. Second, while the open-source CafeOBJ has a proven track-record in the verification of security protocols [19, 18, 17, 16, 14, 15,

20], our proof serves once more as a case study to show that theorem proving in CafeOBJ scales well beyond simple academic problems to real-world scenarios. Third, to our best knowledge, we are the first to model a Diffie-Hellman key-exchange to such detail in CafeOBJ. This might serve as a foundation for analysing other DH-based protocols.

The source code of the proof is available under a free license from the author upon request.

The structure of this paper is as follows: In Section 2 we introduce the PACE protocol. A very brief recapitulation of modeling OTS's in CafeOBJ, and proving their invariants is given in Section 3. We provide an abstract version of PACE show how to model and prove key secrecy, and reflect on lessons learned in Section 4 Properties of PACE have already been analysed in the proprietary VSE-tool [11]. In Section 5 we relate our work to that proof. Last, we conclude our presentation in Section 6 and mention future directions. In our presentation we assume familiarity with CafeOBJ.

2 The PACE key agreement protocol

The goal of the PACE key agreement protocol is to establish a secure, authenticated connection between the chip inside a machine readable travel document (MRTD) and a corresponding terminal (PCD). PACE uses a pre-shared low entropy password to derive a high-entropy one using a Diffie-Hellman key exchange [8]. The protocol is versatile in the sense that it allows to use either standard multiplicative groups of integers modulo p or groups based on elliptic curves.

The protocol works as follows: First, it is assumed that a common low entropy password π is known both by the chip and the terminal. Depending on the document type (international travel document, national id-card etc.) and use-case (border control, e-commerce) three solutions exist in practice: 1.) The password is derived from the Machine Readable Zone (MRZ), 2.) the password is derived from a Card Access Number (CAN) specifically printed on the document for this purpose or 3.) the password is derived from a secret personal identification number (PIN) known only to the owner of the document. In all cases, the password is stored on the chip in a protected way. To read out data on the chip, the MRZ is optically read by, or the CAN or the PIN is entered manually into the terminal.

In the next step, the chip sends both a random nonce s encrypted by a symmetric cipher with π and the domain parameter D_{PICC} for the group operation to the terminal. Using a *mapping function* and the domain parameter, the nonce s is mapped to some generator g of the group $\langle g \rangle$. Both the terminal and MRTD chip chose another nonce x resp. y and compute exponents, i.e. the group operation is applied with the nonce together with the generator to derive g^x resp. g^y. These are then shared, and a key $K = (g^x)^y = (g^y)^x$ and MAC and Session-Keys are derived. Knowledge of the sent exponents and the key is verified by

MRTD Chip (PICC)		Terminal (PCD)
	shared password π	
choose *nonce* $s \leftarrow \mathbb{Z}_q$ static domain parameter D_{PICC}		
$z = \mathbf{enc}(\mathcal{H}(\pi), s)$	$\xrightarrow{D_{\mathrm{PICC}}, z}$	
		$s = \mathbf{dec}(\mathcal{H}(\pi), z)$
$g = \mathbf{map}(D_{\mathrm{PICC}}, s)$		$g = \mathbf{map}(D_{\mathrm{PICC}}, s)$
choose $x \leftarrow \mathbb{Z}_q^*$		choose $y \leftarrow \mathbb{Z}_q^*$
$h_1 = g^x$		$h_2 = g^y$
	$\xrightarrow{h_1}$	
	$\xleftarrow{h_2}$	
abort, if $h_2 \notin \langle g \rangle$ or $h_1 \doteq h_2$		abort, if $h_2 \notin \langle g \rangle$ or $h_2 \doteq h_1$
$K = h_2^x = (g^y)^x$		$K = h_1^y = (g^x)^y$
$K_{\mathrm{MAC}} = \mathcal{H}(K\|\|1)$		$K_{\mathrm{MAC}} = \mathcal{H}(K\|\|1)$
$K_{\mathrm{ENC}} = \mathcal{H}(K\|\|2)$		$K_{\mathrm{ENC}} = \mathcal{H}(K\|\|2)$
$T_{\mathrm{PICC}} = \mathbf{mac}(K_{\mathrm{MAC}}, h_2)$		$T_{\mathrm{PCD}} = \mathbf{mac}(K_{\mathrm{MAC}}, h_1)$
	$\xrightarrow{T_{\mathrm{PICC}}}$	
	$\xleftarrow{T_{\mathrm{PCD}}}$	
abort, if $T_{\mathrm{PCD}} \neq \mathbf{mac}(K_{\mathrm{MAC}}, h_1)$		abort, if $T_{\mathrm{PICC}} \neq \mathbf{mac}(K_{\mathrm{MAC}}, h_2)$

Fig. 1. The PACE protocol.

exchanging MAC-tokens. See Figure 1 for a brief overview of the protocol. For more detailed specifications we refer to [3].

3 OTS, CafeOBJ and invariant-proving

Our proof approach is based on Observational Transition Systems (OTS's). For precise definitions and an introduction to OTS's we refer to [17]. For a more in-depth treatment and the theoretical foundations of CafeOBJ we refer to [7]. Here we only briefly recapitulate how OTS's are modelled in CafeOBJ in order to give an intuition of the overall proof approach and proof structure. An OTS is a triple of a set of observable values, a set of initial states, and a set of of conditional transition rules. A protocol can be modelled as an OTS, where each state of the protocol can be *observed*, and the effect of a state change on these observations is described by *transitions*. An *invariant* is a property that holds (is observable) in all states reachable from the initial ones. We have the following correspondence between an OTS and CafeOBJ:

– The state space is modelled as a hidden sort H.

– A data types D is described in order-sorted algebra with visible sort V.
– An observation is modelled as a CafeOBJ behavioural operator:

```
bop o : H V1 V2 ... VN -> V
```

Here `V1,...,VN` and `V` are visible sorts corresponding to data types $D_1, \ldots, D_n$, and `H` is the hidden sort representing the state space.
– A transition is also modelled as a CafeOBJ behavioural operator:

```
bop t : H V1 V2 ... VM -> H
```

The first argument of `t` refers to the current state. The operator `t` — identified by the indices `V1 ... VM` — maps the current state to another state in the state space. How this transition operator affects the state space in particular, is defined in CafeOBJ in the following manner:

```
ceq o(t(X,Y1,...,YM),Z1,...,ZN) =
    changeval(X,Y1,...,YM,Z1,...ZN)
        if effective-condition(X,Y1,...,YM,Z1,...,ZN) .

ceq t(X,Y1,...,YM) = X
        if not effective-condition(X,Y1,...,YM,Z1,...,ZN) .
```

Here `changeval` is the operation that changes values of the observation to the one of the successor state, and `effective-condition` evaluates whether the condition to apply the transition is met in the current state. If the observed value does never change when applying the transition we can combine the above simply to: `eq o(t(X,Y1,...,YM),Z1,...,ZN) = o(X,Y1,...,YM)`.

Proof Scores A proof score of an invariant consists of two parts: First, the induction hypothesis w.r.t. the predicate in the initial state is shown. Then the induction step follows. For each invariant $\text{pred}_i(s, \mathbf{x})$ we define a corresponding operator and an equation

```
op invI : H V1 V2 ... VN -> Bool .
eq invI(S,X1,...,XN) = ... .
```

These definitions are grouped in a module `INV`. In the definitions of visible sorts in our model, we also define a constant `init` denoting an arbitrary initial state. Then to prove $\text{pred}_i(s, \mathbf{x})$ we simply open the module `INV`, fix arbitrary objects `v1,...,vN` for the visible sorts `V1,...,VN` and issue a reduce command w.r.t. the initial state `red invI(init,v1,...,vN)`.

For the induction step we have to show that if $\text{pred}_i(s, \mathbf{x})$ holds in state s, then it also holds in any possible next state s'. For each predicate we fix arbitrary states s and s', define an operator of form `op istepI : V1 V2 ... VN -> Bool` and an equation

```
eq istepI(X,Y,...) = invI(s,X1,,...,XN) implies invI(s',X1,...,XN) .
```

These definitions are grouped in a module `ISTEP`. To prove the induction step it then suffices to open the module, fix arbitrary objects `v1,...,vN` for the visible sorts and issue a reduce command `red istepI(v1,...,vN)`.

Lemmata Quite often the induction step cannot be shown directly, since the induction hypothesis is too weak. Then we need to apply a lemma. Let `invJ` be a predicate with free variables of visible sorts `E1,...,EK`, and let `e1,...,eK` denote either free variables of, or expressions (i.e. terms) of these sorts. We can strengthen the induction hypothesis by augmenting `invJ` in state `s`, i.e. by issuing `red invJ(s,e1,...eK) implies istepI(v1,...,vN)`. One advantage in the observational approach is that we can use `invJ` to strengthen the induction step in the proof of `invI` and vice-versa.

Case Analysis Another possibility is to apply case analysis. Suppose for example `v1` is defined as an arbitrary object, and we consider the case that either `v1` is constructed by the constructor `f` applied to some other object `vC`, or that this is not the case. Then the induction step is done in two steps: We first open `ISTEP`, declare `v1 = f(vC)`, and reduce `red istepI(v1,...)`. Then we do the same again, but declare `(v1 = f(vC)) = false` before reducing. Clearly we have exhaustively considered all possible cases, since it is always true that:

```
(v1 = f(vC)) or (not (v1 = f(vC)))
```

Of course it is possible to strengthen the induction hypothesis by more than one predicate, to stack case analysis, and to combine lemma application with case analysis.

4 Modelling PACE in CafeOBJ

Our proof is in the Dolev-Yao model [9]. We assume that principals interact with each other by sending messages, and distinguish between honest principals who behave according to the protocol, and malicious ones that fake and forge messages. The malicious principals are modelled as the most general intruder. Moreover we make the following assumption:

1. Cryptographic primitives are sound. Random nonces are unique and cannot be guessed, encrypted messages can only be decoded by knowing the correct key, hashes are one-way and there are no collisions, and two message authentication codes are the same only if generated from the same message with the same key.
2. The intruder can glean any public information (i.e. messages, ciphers etc.) that is sent in the network.
3. The intruder can send two kinds of messages: He can use ciphers based on cryptographic primitives from existing messages as black boxes to send new fake messages, and he can use eavesdropped information to generate new messages from scratch. But as noted above, he cannot eavesdrop information from ciphers based on cryptographic primitives without knowing the corresponding keys or passwords.

4.1 An abstract version of PACE

To abstract away from implementation-dependent information and those that cannot be captured in the Dolev-Yao model anyway, we define the following abstract version of the PACE protocol.

$$
\begin{array}{lll}
\text{Message 1}: & p \to q: & \mathbf{enc}_{\pi}(n_s, D) \\
\text{Message 2}: & p \to q: & *(n_a, G) \\
\text{Message 3}: & q \to p: & *(n_b, G) \\
\text{Message 4}: & p \to q: & \mathbf{mac}(\mathcal{H}(*(n_a, *(n_b, G))), *(n_b, G), D) \\
\text{Message 5}: & q \to p: & \mathbf{mac}(\mathcal{H}(*(n_b, *(n_a, G))), *(n_a, G), D)
\end{array}
$$

We assume that a run of PACE is conducted by exchanging five messages. In the first step, a message is sent from a participant p to another one q. The message encrypts a random nonce n_s with the shared password π, with attached static domain parameters D. Next, p maps the nonce n_s from the first message with the domain parameters to a group generator G. Then p chooses a random nonce n_a, applies the group operator $*$ to both n_a and G and sends the result $*(n_a, G)$ to q. In a similar manner, q chooses a random nonce n_b and sends $*(n_b, G)$ to p. Next, p computes the key $\mathcal{H}(*(n_a, *(n_b, G)))$. He then sends a message authentication code — encoded with that key — with the received exponent $*(n_b, G)$ and domain parameters D to q, in order to verify knowledge of both the received exponent and the generated key. Principal q does the same in reverse, and the common key $\mathcal{H}(*(n_a, *(n_b, G)))$ is used from now on to exchange encrypted messages.

4.2 Basic Data Types

We use the following algebraic data-types, i.e. visible sorts and corresponding constructors:

- `Principal` denotes both honest and malicious participants in the network.
- `Random` denotes random nonces. We suppose that random nonces are unique and unguessable.
- `Dompar` denotes the static domain parameters of PACE. Used domain parameters are not secret and known to every principal.
- `Mappoint` denotes a group generator. The constructor `maptopoint` of data type `Mappoint` takes as input a random nonce and static domain parameters and returns a group generator. We suppose that `maptopoint` is a one-way function.
- `Expo` denotes an exponent of the form g^x, where the group generator g is generated by `maptopoint` using a random nonce and domain parameters as input.
- `Hash` denotes keys — we suppose that hashing is our key derivation function. The constructor `hash` takes as input a random nonce and an exponent and returns a key.

– `Cipher1` denotes the cipher resulting from a symmetric encryption. Its constructor `enc` takes as input a random nonce and static domain parameters. We assume implicitly that a `Cipher1` is encoded with the shared password π in the following way: Given a `Cipher1`, every principal is able reconstruct the static domain parameters. But only if he knows the shared password π, he is able to decode the random nonce.
– `Cipher3` denotes message authentication codes. The constructor `mac` takes as input a hash, an exponent and domain parameters.

We define three sorts and data types for the messages in Section 4: Message 1 of Section 4 is of type `Message1`, messages 2 and 3 are of type `Message2`, and messages 4 and 5 are of type `Message3`. Here, `Message1` is a `Cipher1` attached with meta-information describing the creator, the (seemingly) sender, and the receiver of a message. Similar, a `Message3` is a `Cipher3` attached with corresponding meta-information. The data type `Message2` is constructed by attaching meta-information to an exponent. Moreover for the definition of the data structures we point to two design decisions:

Modelling of the shared password PACE assumes a fixed shared password π known among honest principals. Knowledge of the password is modelled by a predicate `knowspi` where we set `knowspi(intruder) = false`. We do not introduce a specific data type for decryption of messages of type 1 but simply distinguish between messages that are created by an honest principal who does know π and the intruder, who does not. This reduces the number of data types and thereby yields less complexity in the case analysis of proofs.

Equality of hashes We define the equality operator `_=_` for `hashes` as

```
eq (H1 = H2)
   = ((rand(H1) = rand(H2) and expo(H1) = expo(H2)) or
      (rand(H1) = rand(expo(H2)) and rand(H2) = rand(expo(H1))
       and point(expo(H1)) = point(expo(H2)))) .
```

in order to capture the properties of the group operator, namely that $(g^x)^y = (g^y)^x$. When defining equality among message ciphers one might be tempted to define equality for two ciphers3's `C1` and `C2` recursively as:

```
eq (C1 = C2) = (hash(C1) = hash(C2) and expo(C1) = expo(C2)
            and dpar(C1) = dpar(C2)) .
```

This has the awkward consequence that messages can no longer uniquely be identified: When a principal sends a message of type 3, implicitly *two* messages are added to the network, one w.r.t. each case of equality of the hash. This makes reasoning in the induction steps quite unintuitive: Suppose we have a cipher3 `c = mac(hash(r1,expo(r2,...),...)` and perform a case analysis of a cipher3 `c3` w.r.t. to `c`. Then considering the cases `c3 = c` and `(c3 = c) = false` is not exhaustive, as it misses the case `c3 = mac(hash(r2,expo(r1,...),...)`. In our

modelling we define equality of cipher3's as syntactic equality of normals forms, and formulate our theorems accordingly when we refer to multiple cipher3's with the same hash.

4.3 Protocol Modelling

CafeOBJ is based on first order equational rewriting. It therefore lacks higher order constructs. In order to collect all sent messages, all generated random nonces, and other information, we reuse the following definition of a *multiset* on an abstract level from [17], and later use this as a parametrized module to adapt this to multisets containing the data-types defined in the previous section.

```
mod* SOUP (D :: EQTRIV) principal-sort Soup {
  [Elt.D < Soup]
  op empty : -> Soup {constr}
  op _ _ : Soup Soup -> Soup {constr assoc comm id: empty}
  op _\in_ : Elt.D Soup -> Bool
  var S : Soup
  vars E1 E2 : Elt.D
  eq E1 \in empty = false .
  eq E1 \in (E2 S) = (E1 = E2) or E1 \in S .
}
```

Here the operator `\in` defines membership in the multiset, and a space defines insertion. To collect all random nonces for example, we can define an observation `bop rands : System -> RandSoup` that takes as input a state, and returns as the observation a soup of random nonces. Given a random nonce `r` and a state `s`, we can test membership by `r \in rands(s)`, and — for example describing the effects of a transition — insert `r` in the multiset by `r rands(s)`. Observations and transitions are defined as follows:

```
-- observations
bop network  : System -> Network
bop rands    : System -> RandSoup
bop hashes   : System -> HashSoup
bop randsi   : System -> RandSoup
bop expos    : System -> ExpoSoup
bop cipher1s : System -> Cipher1Soup
bop cipher3s : System -> Cipher3Soup
-- transitions
bop sdm1  : System Principal Principal Random Dompar           -> System
bop sdm2  : System Principal Principal Random Message1         -> System
bop sdm3  : System Principal Principal Message1 Message2 Message2
                                                               -> System
-- faking and forging messages based on the gleaned info
bop fkm11 : System Principal Principal Cipher1                 -> System
bop fkm12 : System Principal Principal Random Dompar           -> System
bop fkm21 : System Principal Principal Expo                    -> System
```

```
bop fkm22 : System Principal Principal Random Random Dompar    -> System
bop fkm31 : System Principal Principal Cipher3                 -> System
bop fkm32 : System Principal Principal Random Expo Expo Dompar -> System
```

We use seven observers to collect information:

- `network` returns a multiset of *all* messages that have been sent so far.
- `rands` returns a multiset containing *all* random nonces that have been generated so far.
- `hashes` returns all *keys* resulting from the PACE protocol that have been gleaned or self-generated by the *intruder*. The name stems from the fact that we consider `hash` to be the key derivation function.
- `randsi` contains all *random nonces* gleaned or self-generated by the *intruder*.
- `expos` contains all exponents that have been inserted in the network and
- `cipher1s` and `cipher3s` collect *all* ciphertexts of messages of type 1 and messages of type 3 (i.e. mac-tokens).

The transitions `sdm1`, `sdm2`, and `sdm3` describe state transitions and their effects on observations when an honest principal sends a message. Therefore the conditions on when these transitions are effective capture precisely the behaviour of an honest principal. For example `sdm1` is defined as:

```
eq c-sdm1(S,P,Q,R,D) = not(R \in rands(S)) .
ceq network(sdm1(S,P,Q,R,D))  = me1(P,P,Q,enc(R,D)) network(S)
    if c-sdm1(S,P,Q,R,D) .
```

Here the effective condition for an honest principal to send a message of type 1 is that the random nonce used is fresh: It must not be contained in the multiset of already used nonces.

The transitions `fkmXY` describe state transitions and their effects on observations when the intruder generates messages. Here we distinguish two cases: 1.) the intruder fakes an existing message by changing its source and destination (`fkmX1`) and 2.) the intruder injects a new message in the network using information available to him (`fkmX2`). Therefore the effective conditions for these transitions are usually more lax than the ones for `sdmX`. For example the condition to fake a message of type 1

```
eq c-fkm11(S,P,Q,C1) = C1 \in cipher1s(S) .
```

is just that a cipher1 exists in the network. The intruder can then inject the message `me1(intruder,P,Q,C1)` with arbitrary source `P` and destination `Q`. Note that the meta information denoting the creator of the message cannot be altered by the intruder.

An example for the second case is the condition to construct an arbitrary new message of type 1

```
eq c-fkm12(S,P,Q,R,D) = (not (R \in rands(S))) or (R \in randsi(S)) .
```

Here the intruder can choose to either use a fresh random nonce, or one that he has gleaned or generated in an earlier state. He then injects the message `me1(intruder,P,Q,enc(R,D))` into the network.

4.4 Proving Key-Secrecy

We show key secrecy in the following sense: Suppose we take the perspective of an honest principal, i.e. we are either the MRTD or the terminal, and we behave according to protocol. In particular we assume

1. We have either sent a `Message1` with a nonce encrypted with the shared password π and domain parameters *or* we have received a `Message1` from a principal who knows π and decrypted it and
2. we constructed a generator of the group with the nonce and the domain parameters from the above message, used the generator together with a fresh nonce to create an exponent, and sent it to the other party and
3. we *seemingly* (we do not know who created the message) received an exponent back from that other party and
4. we *seemingly* received a MAC-token that, using our secret nonce together with the received exponent as a key, validates that the other party knows our sent exponent and the domain parameters.

Then the resulting key must *never* be known to the intruder. This can be almost verbatim translated into our main theorem:

```
eq inv900(S,M1,M21,M22,M3,P,Q) =
  (M1 \in network(S) and M21 \in network(S)
   and M22 \in network(S) and M3 \in network(S)
   and sender(M3) = Q and receiver(M3) = P
   and creator(M21) = P and sender(M21) = P and receiver(M21) = Q
   and sender(M22) = Q and receiver(M22) = P
   and (not (creator(M21) = creator(M3)))
   and (not (P = Q)) and knowspi(P)
   and ((sender(M1) = P and creator(M1) = P and receiver(M1) = Q) or
        (sender(M1) = Q and receiver(M1) = P and knowspi(creator(M1))))
   and expo(M21) = expo(cipher3(M3))
   and dpar(cipher1(M1)) = dpar(point(expo(M21)))
   and rand(cipher1(M1)) = rand(point(expo(M21)))
   and dpar(cipher1(M1)) = dpar(cipher3(M3))
   and hash(cipher3(M3)) = hash(rand(expo(M21)),expo(M22)))
 implies
   not (hash(cipher3(M3)) \in hashes(S)) .
```

Collecting Hashes In our modeling a `hash` (a key) is only added to the gleaned set of hashes `hashes(S)` of the intruder, if a transition `fkm32` occurs. One might argue that with our definition, proving absence of the hash in the multiset `hashes(S)` not sufficient, since the following situation might occur: Let `m3` be a message of type 3, `s` be a state, `r1`, `r2` and `r3` all gleaned random nonces included in `randsi(s)`, `d` domain parameters known by the intruder, and `h` the hash `hash(r1,expo(r2,maptopoint(r3,d)))` such that: `hash(cipher3(m3)) = h`, but `h` has not been used in a message of type 3 and is therefore not contained in `hashes(S)`. This would mean that invariant `inv900` does not capture true

key secrecy. Note however that in such a case, the intruder can always add `h` to `hashes(S)` by performing the following steps:

1. He inserts a `me2(intruder,p,expo(r2,maptopoint(r3,d)))` into the network using transition `fkm22` with an arbitrary sender and receiver `p`. By definition of the transition `fkm22` that exponent is included in `expos(s')` of the next state `s'`.
2. He uses that exponent to send the following message by transition `fkm32` to the network: `me3(hash(r1,expo(r2,maptopoint(r3,d))),e,d)`. Here `e` is an arbitrary exponent and `d` arbitrary domain parameters. Then by definition of `fkm32` the hash `h` is included in `hashes(s'')` of the next state.

In our modeling we have chosen to bind the addition of hashes to the multiset `hashes(s)` to sending a message by transition `fkm32` as a way to reduce the complexity of the case analysis in the proof — in our modeling, the most complex case analysis takes only place in the induction step w.r.t. `fkm32`.

Application of Lemmata and Case Analysis To prove key secrecy we need additional invariants. Central to strengthen the induction hypothesis for `istep900` is the invariant that the assumptions of `inv900` imply that both participants have implicitly agreed upon the same generator g, which itself depends on the nonce exchanged in the first message. For brevity suppose that `assump(S,M1,M21,M22,P,Q)` is a predicate that denotes truth of the assumptions of invariant `inv900` above. The invariant can then be expressed as:

```
eq inv800(S,M1,M21,M22,M3,P,Q) = assump(S,M1,M21,M22,P,Q)
  implies rand(point(expo(M22))) = rand(point(expo(M21))) .
```

We illustrate how such a lemma is used in the proof together with case analysis, albeit for a simpler invariant. We make frequent use of the following invariant as a lemma for others. It states that if we are in a state `S`, and a `M1` of type `Message1` is in the network, then the random nonce of `M1` has been used and is thus included in the collection of all random nonces `rands(S)`.

```
eq inv300(S,M1) = M1 \in network(S)
                  implies rand(cipher1(M1)) \in rands(S) .
```

We prove `inv300` inductively on the number of transitions. In the case of transition `fkm11` we perform case analysis w.r.t. its effective condition:

```
(c-fkm11(s,p10,q10,c11) = false) or (c-fkm11(s,p10,q10,c11) = true)
```

Here `p10` and `q10` denote arbitrary principals, and `c11` denotes an arbitrary `cipher1`. For the first case, the proof directly succeeds:

```
open ISTEP
 ops p10 q10 : -> Principal .
 op m10 : -> Message1 .
 op c11 : -> Cipher1 .
```

```
 eq c-fkm11(s,p10,q10,c10) = false .
 eq s' = fkm11(s,p10,q10,r10,d10) .
 red istep300(m10) .
close
```

For the second case `c-fkm11(s,p10,q10,c11) = true`, we replace the term with its definition `c11 \in cipher1s(s) = true` and perform another case analysis w.r.t. the equality `m10 = me1(intruder,p10,q10,c11)`.

```
open ISTEP
 ops p10 q10 : -> Principal .
 ops m10 : -> Message1 .
 op c11 : -> Cipher1 .
 eq c11 \in cipher1s(s) = true .
 eq m10 = me1(intruder,p10,q10,c11) .
 eq s' = fkm11(s,p10,q10,c11) .
 ***
close
```

If we directly try to proof the induction step by reducing `red istep300(m10)` inserted at `***`, CafeOBJ outputs

```
rand(c11) \in rands(s) xor
  me1(intruder,p10,q10,c11) \in network(s) xor ...
```

This indicates that if `me1(intruder,p10,q10,c11)` is not already included in and thus inserted in the network as a result of the transition `fkm11`, then `rand(c11) \in rands(s)` must be true for the induction step to hold. Therefore the induction hypothesis needs to be strengthened. We do so by introducing yet another invariant `inv150`, which states that if a `cipher1` is in the network, than its random nonce is included in the set of all used random nonces.

```
eq inv150(S,C1) = C1 \in cipher1s(S) implies rand(C1) \in rands(S) .
```

And indeed, applying `inv150` as a lemma at `***` by inserting

```
red inv150(s,c11) implies istep300(m10)
```

successfully finishes the induction step.

Lessons Learned From the perspective of applying the OTS/CafeOBJ method in practice, we have found the following to be the two major advantages of that approach: First, despite the lack of quantors, the verbosity of CafeOBJ allows for a very compact formalization of the protocol itself, and the invariants we want to prove. In fact, in our formalization, only 505 lines are used for PACE itself, and 445 lines are for definition of invariants as modules `INV` and `ISTEP`.

Second, the approach does not force oneself into a strict sequential or backward approach when proving. This is supported by the fact that we can prove each conjunct of a large formula individually in a way where each predicate may be used to strengthen the induction hypothesis of another one without having to

worry about mutual dependencies. In practice this yields a lot of freedom when building up the proof. For example we can first focus on very basic and rather technical invariants, such as `inv300` as mentioned above. Such basic properties can then later be applied as lemmata, simplifying the proof of more complex invariants.

With more complex invariants, such as our main theorem `inv900`, it turned out to be useful to directly apply a proof attempt. Such a complex proof can usually not be shown directly, however the stuck proof attempt is helpful to discover needed lemmata. When one discovers a needed lemma, it is advantageous to have the ability to quickly conjecture an invariant without proving it, and to try whether that invariant helps to finish the more complex proof. Only after testing whether the more complex one can be proven, we need to focus on the lemma. Quickly jumping between such a backtracking, and the above mentioned forward based approach gives the user a large amount of flexibility.

Conducting a proof attempt implicitly facilitates lemma generation. When CafeOBJ outputs the result of a reduction different from `true`, the reduced term can be used directly to apply case analysis. In such case analysis the vast amount of cases will be proven trivially. This leads us to the difficult part of the proof, which is then characterized as a very specific case with several assumptions, say $a_1, a_2, a_3, \ldots$. If the proof is stuck here, then $\neg(a_1 \wedge a_2 \wedge a_3 \wedge \ldots)$ is a natural candidate lemma. Even though in practice this candidate lemma needs to be generalized or slightly modified, this often gives the operator a hint to what major property is needed to successfully conduct the proof.

The main hindrance we found with the OTS/CafeOBJ method is related to performance. Suppose we are proving an invariant of the form $a_1 \wedge a_2 \ldots a_n \implies b$, such as `inv900`. A direct proof attempt often does not terminate, due to the amount of branching. To get a terminating result, one can make a trivial case analysis w.r.t. a_i, e.g. distinguish the case for $\neg a_1$, for $a_1 \wedge \neg a_2$, and so on, to finally reach the case for $a_1 \wedge \ldots \wedge a_n$. Even then sometimes a proof attempt does not terminate, so additional (trivial) assumptions and corresponding cases have to be added. Almost all cases are trivial – it is obvious that in the case with the assumption $\neg a_1$ the above invariant holds – but lead to a tedious copy and paste approach and unnecessarily blow up the size of the proofs. Our proof score for example consists of 38427 lines, of which the vast majority are for such trivial cases.

All in all, we have proved 40 invariants of our formalization of PACE. The verification of all invariants together takes approximately two hours and eight minutes on an Intel Core i7-3520M @ 2.9 Ghz.

5 Related Work

Using inductive theorem proving to verify properties of security protocols was first pioneered in [21] using Isabelle/HOL [13]. An inductive verification [4] of the PACE protocol has already been conducted in the verification support environment (VSE) [11]. VSE has been developed by a consortium of German

universities and industry to provide a tool to meet industry needs for the development of highly trustworthy systems, with a high degree of automation in mind. The proof for PACE for example claims a 70 percent level of automation. The proof of PACE includes not only key secrecy, but also mutual authentication and perfect forward secrecy, but is not publicly available. Even if it would however, independent verification might be difficult as VSE is currently not licensed as open-source. A non-mechanical proof for security in the sense of Abdalla, Fouque and Pointcheval [1] has been given in [2]. In [5] attempts are made to merge these two analysis'.

CafeOBJ touts itself as an industry strength algebraic specification language, licensed under the GNU general public license. In general, its observational approach using rewriting without quantifiers is incomparable to the inductive approach based on first order logic of VSE. Our proof of key-secrecy has been developed independently from [4]. Aside from rewriting, our proof is mostly manual. Nevertheless, as mentioned in the previous section, the OTS/CafeOBJ method implicitly supports the user in discovering lemmata, and thus significantly simplifies the complexity of a proof.

6 Conclusion and Future Work

We have successfully verified key secrecy in CafeOBJ. This not only facilitates trust in the PACE protocol, but also represents one more case-study that shows that the OTS/CafeOBJ approach scales well beyond toy-examples like NSPK to real-world scenarios. Also, the PACE proof can serve as a guide on how to model a DH-key exchange in CafeOBJ. Key-Secrecy however, is only one important property of PACE. We plan to extend the proof to mutual authentication and perfect forward secrecy.

The most tedious part in writing proof scores is the lack of automation and the need to write down large amounts of redundant information when performing case analysis. As mentioned in the previous section, this often leads to a "copy-and-paste" approach. Very recently, tool support has been increased significantly [10], and we plan to lift our proof to that platform.

We thank the anonymous reviewers for their helpful comments.

References

1. Abdalla, M., Fouque, P.A., Pointcheval, D.: Password-based authenticated key exchange in the three-party setting. In: Proc. 8th PKC. LNCS, vol. 3386, pp. 65–84 (2005)
2. Bender, J., Fischlin, M., Kügler, D.: Security analysis of the pace key-agreement protocol. In: ISC. LNCS, vol. 5735, pp. 33–48 (2009)
3. BSI: Advanced security mechanisms for machine readable travel documents, v2.11. Tech. rep., Bundesamt für Sicherheit in der Informationstechnik (2012)
4. Cheikhrouhou, L., Stephan, W.: Meilensteinreport: Inductive verification of pace. Tech. rep., Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (2010)

5. Cheikhrouhou, L., Stephan, W., Dagdelen, Ö., Fischlin, M., Ullmann, M.: Merging the cryptographic security analysis and the algebraic-logic security proof of pace. In: Sicherheit. Lecture Notes in Informatics, vol. 195, pp. 83–94. Gesellschaft für Informatik (2012)
6. Clark, J., Jacob, J.: A survey of authentication protocol literature: Version 1.0 (1997)
7. Diaconescu, R., Futatsugi, K.: CafeOBJ Report: The language, proof techniques and methodologies for object-oriented algebraic specification, AMAST Series in Computing, vol. 6. World Scientific (1998)
8. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976)
9. Dolev, D., Yao, A.C.C.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–207 (1983)
10. Găină, D., Zhang, M., Chiba, Y., Arimoto, Y.: Constructor-based inductive theorem prover. In: Proc. 5th CALCO. Lecture Notes in Computer Science, vol. 8089, pp. 328–333 (2013)
11. Koch, F.A., Ullmann, M., Wittmann, S.: Verification support environment. In: Proc. 8th CAV. LNCS, vol. 1102, pp. 454–457 (1996)
12. Lowe, G.: An attack on the needham-schroeder public-key authentication protocol. Inf. Process. Lett. 56(3), 131–133 (1995)
13. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
14. Ogata, K., Futatsugi, K.: Formal analysis of the ikp electronic payment protocols. In: Proc. 1st ISSS. LNCS, vol. 2609, pp. 441–460 (2002)
15. Ogata, K., Futatsugi, K.: Rewriting-based verification of authentication protocols. Electr. Notes Theor. Comput. Sci. 71, 208–222 (2002)
16. Ogata, K., Futatsugi, K.: Flaw and modification of the ikp electronic payment protocols. Inf. Process. Lett. 86(2), 57–62 (2003)
17. Ogata, K., Futatsugi, K.: Proof scores in the OTS/CafeOBJ method. In: Proc. 6th FMOODS 2003. LNCS, vol. 2884, pp. 170–184 (2003)
18. Ogata, K., Futatsugi, K.: Equational approach to formal analysis of tls. In: Proc. 25th ICDCS. pp. 795–804. IEEE Computer Society (2005)
19. Ogata, K., Futatsugi, K.: Proof score approach to analysis of electronic commerce protocols. International Journal of Software Engineering and Knowledge Engineering 20(2), 253–287 (2010)
20. Ouranos, I., Ogata, K., Stefaneas, P.S.: Formal analysis of tesla protocol in the timed ots/cafeobj method. In: Proc. 5th ISoLA. LNCS, vol. 7610, pp. 126–142 (2012)
21. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security 6(1-2), 85–128 (1998)

Formal Modeling and Analysis of Cassandra in Maude

Si Liu, Muntasir Raihan Rahman, Stephen Skeirik,
Indranil Gupta, and José Meseguer

University of Illinois at Urbana-Champaign

Distributed key-value (e.g., Cassandra [2], RIAK [1]) storage systems are increasingly being used to store and query data in today's industrial deployments. Many diverse companies and organizations are moving away from traditional strongly consistent databases and are instead using key-value/NoSQL stores in order to store huge data sets and tackle an increasing number of users.

Distributed key-value stores typically replicate data on multiple servers for greater availability in the presence of failures. Since any of the replicas storing the data can now respond to client read requests, it becomes costly to always keep all the replicas synchronized. This creates a tension between consistency (keeping all replicas synchronized) and availability (replying to clients quickly), especially when the network is partitioned [3]. Whereas traditional databases prefer consistency over availability, distributed key-value stores risk exposing stale data to clients to remain highly available. This approach was popularized by the Dynamo [5] key-value store architecture from Amazon. Cassandra [2] is an open-source distributed key-value store which closely follows the Dynamo architecture. Many large scale Internet service companies like Netflix, IBM, HP, Facebook, Spotify, and PBS Kids rely heavily on the Cassandra key-value storage system.

Weakly consistent key-value stores like Cassandra typically employ many complex design decisions that can impact the consistency and availability guarantees offered to the clients. Therefore, there is an urgent need to develop formal models for specifying these design decisions and formal methods for reasoning about the impact of these design choices on specified consistency (correctness) and availability (performance) guarantees. Equipped with such a formal model, it becomes very convenient to explore new design choices and formally compare them with existing design decisions.

Today there are two main approaches for verifying consistency models for distributed key-value stores. First, we can run a given key-value store under a particular environment, and audit the read/write operation logs to check for consistency violations. Second, we can analyze the algorithms used by the key-value store to ensure consistency. However, the first approach is not guaranteed to find all violations of the consistency model. Using the second approach is time consuming and needs to be repeated for every system with different implementations of the underlying algorithms for guaranteeing consistency.

In this paper, we present a formal executable model of Cassandra. Our model is specified in Maude [4], a modeling language based on rewriting logic. Rewriting logic was proposed in the early nineties as a unified model for concurrency in

which several well-known models of concurrent and distributed systems can be represented. Our formal Maude model for Cassandra includes component models for data partitioning strategies, consistency levels, read repair, and timestamp policies for ordering multiple versions of data. These component models that represent various design decisions employed by Cassandra are specified using Maude rewrite rules. The built-in *linear temporal logic* (LTL) model checker of Maude can exhaustively search consistency violations for all possible delay distributions and ordering of messages between clients, coordinators, and servers storing replicas of the same data in the system.

Our formal model can also be easily used to specify and analyze new design choices for a particular component of the key-value store. As a concrete example, we present a new read processing strategy that is timestamp agnostic. Since our Maude based Cassandra model is executable, we have been able to formally compare consistency and availability behavior for both the existing timestamp-based strategy and our new timestamp-agnostic strategy. Our Maude specification is fewer than 1000 lines of code, compared to the Cassandra code base which is over a million lines of code. As a result our formal approach for testing new design choices is faster and more accurate than existing approaches that modify huge code bases.

Concretely the technical contributions of this paper are:

- We present, to the best of our knowledge for the first time, a formal executable model for the Cassandra key-value store using Maude rewriting logic. Our model can be used to exhaustively search for violations of both strong and eventual consistency in Cassandra.
- We present a new timestamp agnostic read processing strategy for Cassandra using our model. We compare this new strategy against Cassandra's timestamp based strategy and find that it has higher consistency at the cost of latency. So the new strategy allows us to choose another point in the consistency-availability trade-off space.

References

1. Basho Riak, `http://basho.com/riak/`
2. Cassandra, `http://cassandra.apache.org/`
3. Brewer, E.A.: Towards robust distributed systems. In: Proc. Symposium on Principles of Distributed Computing (PODC) (2000)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350. Springer (2007)
5. Vogels, W.: Amazon's dynamo. All Things Distributed (October 2007), `http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html`

A Framework for Mobile Ad hoc Networks in Real-Time Maude

Si Liu[1], Peter Csaba Ölveczky[2], and José Meseguer[1]

[1] University of Illinois at Urbana-Champaign
[2] University of Oslo

Abstract. Mobile ad hoc networks (MANETs) are increasingly popular and deployed in a wide range of environments. However, it is challenging to formally analyze a MANET, both because there are few reasonably accurate formal models of mobility, and because the large state space caused by the movements of the nodes renders straightforward model checking hard. In particular, the combination of wireless communication and node movement is subtle and does not seem to have been adequately addressed in previous formal methods work. This paper presents a formal executable and parameterized modeling framework for MANETs in Real-Time Maude that integrates several mobility models and wireless communication. We illustrate the use of our modeling framework with the Ad hoc On-Demand Distance Vector (AODV) routing protocol, which allows us to analyze this protocol under different mobility models.

1 Introduction

A *mobile ad hoc network* (MANET) is a self-configuring network of mobile devices (laptops, smart phones, sensors, etc.) that communicate wirelessly and cooperate to provide the necessary network functionality. Since MANETs can form ad hoc networks without fixed infrastructure, they are supposed to have a wide applicability, for example for providing ad hoc networks for cooperating "smart" cars, for emergency responders during accidents, during natural disasters which may disable fixed infrastructure, in battlefield areas, and so on.

Although many such applications are safety-critical and need formal analysis to ensure their correctness, the formal modeling and analysis of MANETs present a number of challenges that include:

1. The need to model node movement realistically.
2. Modeling communication. There is a subtle interaction between wireless communication, which typically is restricted to distances of between 10 and 100 meters, and node mobility. For example, nodes may move into or out of the sender's transmission range *during* the communication delay; furthermore, the sender may itself move during the communication. Modeling communication in MANETs is therefore challenging for formal languages, which are usually based on fixed communication primitives.

3. Since the communication topology of the network depends on the *locations* of the nodes, such locations must be taken into account in the model. However, this leads to very large state spaces, which makes direct model checking analysis unfeasible: if there are m nodes and n locations, there are n^m different nodes/locations states. A 10×10 grid with four nodes would therefore lead to 100 million states just to capture all nodes and their locations.

As explained in Section 7, we are not aware of any formal model that provides a reasonably detailed model of both mobility and communication in MANETs. Because of its expressiveness and flexibility to define models of communication, Real-Time Maude [20] is a promising language for formally modeling MANETs. In this paper we provide, to the best of our knowledge, the first reasonably detailed formal modeling framework for MANETs. In particular, we formalize

- the most popular models for node mobility, and
- geographically bounded wireless communication, which takes into account the interplay between communication delay and mobility,

in Real-Time Maude. Furthermore, we use object-oriented techniques to make it easy to *compose* our framework with a model of a MANET protocol.

Concerning Challenge 3 above, in this paper we do not develop abstraction techniques for node mobility. Instead, to be able to perform model checking analysis, our model is parametric in aspects such as the possible velocities and directions a node can choose. However, even if a node moves slowly, it may still cover the entire area (and hence contribute to an unmanageable state space) given enough time. Another key feature of Real-Time Maude that makes some meaningful model checking analysis of MANETs possible is therefore *time-bounded* model checking, which allows us to analyze scenarios only up to a certain duration (during which the nodes may not reach most locations). Abstracting the state space caused by node mobility and the need to keep track of node locations is the *sine qua non* for serious model checking of MANETs. The point is that this paper lays the foundations for developing such abstractions by providing a first reasonably detailed formal model of location-aware MANETs.

One of the main tasks of a MANET is to maintain an (ad hoc) network, which means that the network must figure out how to route messages between nodes. In this paper we illustrate the use of our MANETs framework by modeling and analyzing the widely used *Ad hoc On-Demand Distance Vector* [22] (AODV) routing protocol for MANETs developed by the IETF MANET working group.

The rest of this paper is organized as follows. Section 2 gives a background to Real-Time Maude. Section 3 briefly introduces MANETs. Section 4 presents our Real-Time Maude modeling framework for MANETs. Section 5 shows how our framework can be used to model the AODV protocol, and Section 6 explains how that model of AODV can be model checked using Real-Time Maude. Finally, Section 7 discusses related work and Section 8 gives some concluding remarks.

Due to space limitations, we have to omit many details; they are all given in our accompanying longer report [14].

2 Real-Time Maude

Real-Time Maude [20] is a language and tool that extends Maude [5] to support the formal specification and analysis of real-time systems.

Specification. A Real-Time Maude module specifies a *real-time rewrite theory* $(\Sigma, E \cup A, IR, TR)$, where:

- Σ is an algebraic *signature*; that is, a set of declarations of *sorts*, *subsorts*, and *function symbols*.
- $(\Sigma, E \cup A)$ is a *membership equational logic theory* [5], with E a set of possibly conditional equations, and A a set of equational axioms such as associativity, commutativity, and identity. $(\Sigma, E \cup A)$ specifies the system's state space as an algebraic data type, and includes a specification of a sort `Time`.
- IR is a set of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form[1] $[l] : t \longrightarrow t'$ **if** $\bigwedge_{j=1}^{m} cond_j$, where each $cond_j$ is either an equality $u_j = v_j$ or a rewrite $t_j \longrightarrow t'_j$, and l is a *label*. Such a rule specifies an *instantaneous transition* from an instance of t to the corresponding instance of t', *provided* the condition holds.
- TR is a set of *tick rules* $l :$ `{`t`}` $\longrightarrow$ `{`t'`}` `in time` τ **if** $cond$ that advance time in the *entire* state t by τ time units.

A class declaration `class` C `|` $att_1 : s_1, \ldots, att_n : s_n$ declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term `<`$O : C \mid att_1 : val_1, ..., att_n : val_n$`>` of sort `Object`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. A *message* is a term of sort `Msg`.

The state of an object-oriented specification is a term of sort `Configuration`, and is a *multiset* of objects and messages. Multiset union is denoted by an associative and commutative juxtaposition operator, so that rewriting is *multiset rewriting*. For example, the rewrite rule

```
rl [l] :  m(O,w)   < O : C | a1 : x, a2 : O', a3 : z >
       =>
          < O : C | a1 : x + w, a2 : O', a3 : z >   dly(m'(O',x), z) .
```

defines a family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`, the attribute `a1` of object `O` is changed to `x + w`, and a new message `dly(m'(O',x),z)` is generated; this message will become the "ripe" message `m'(O',x)` after `z` time units. Attributes whose values do not change and do not affect the next state of other attributes or messages, such as `a3`, need not be mentioned in a rule. Attributes that are unchanged, such as `a2`, can be omitted from right-hand sides of rules.

A *subclass* inherits all the attributes and rules of its superclasses.

[1] An equational condition $u_i = v_i$ can also be a *matching equation*, written u_i`:=` v_i, which instantiates the variables in u_i to the values that make $u_i = v_i$ hold, if any.

Formal Analysis. In this paper, we only consider Real-Time Maude's *linear temporal logic model checker*, which analyzes whether *each* behavior satisfies a temporal logic formula. *State propositions* are terms of sort `Prop`, and their semantics is defined by equations `ceq` *statePattern* `|=` *prop* `=` *b* `if` *cond*, for *b* a term of sort `Bool`, stating that *prop* evaluates to *b* in states that are instances of *statePattern* when the condition *cond* holds. These equations together define *prop* to hold in all states *t* where *t* `|=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). Real-Time Maude provides both *unbounded* and *time-bounded* LTL model checking. The time-bounded model checking command

`(mc` *t* `|=t` *formula* `in time <=` *timeLimit* `.)`

checks whether the temporal logic formula *formula* holds in all behaviors up to duration *timeLimit* starting from the initial state *t*.

3 Mobility and Communication Delay in MANETs

This section gives an overview of the main mobility models used by researchers on protocol evaluations, and of the per-hop delay in wireless communication.

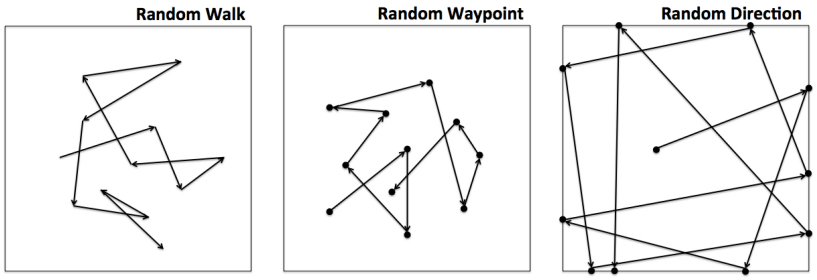


Fig. 1. Motion paths of a mobile node in three mobility models, where a bullet • depicts a pause in the movement.

Mobility Models. Different *mobility patterns* have been proposed to model node mobility in realistic scenarios. In this paper we focus on the following main *entity mobility models* [2], also illustrated in Fig. 1, in which a node's movement is independent of the movements of the other nodes:

– *Random Walk:* Each node moves in "rounds" of fixed durations. A node moves in the same direction and with the same speed throughout one round. At the end of each round, the *new speed* and the *new direction* of a node are randomly chosen, and a new moving round starts.

- *Random Waypoint:* Each node initially pauses for a fixed duration. When a pause ends, a node randomly chooses a *new destination* and a *new speed*, and then travels to that destination at the chosen speed. After arriving, the node again pauses before a new moving round starts.
- *Random Direction:* Each mobile node chooses a *random direction*, along which it travels until reaching the border of the sensing area. When a node arrives at the border, the node pauses for a given time, and then randomly selects a new direction and starts to move in that direction.

Communication Delay. To understand how node movement affects wireless communication, we must understand the messaging delays in wireless communication. In a typical wireless transmit/receive process, the per-hop communication delay from a transmitter to a receiver consists of the following five phases [24]:

Delay Factor	**Description**
Sender Processing Delay	The time elapsed on the sender side from the moment a message timestamp is taken to the point the message is buffered in the device.
Media Access Delay	The time for a message to stay in the radio device buffer; e.g., in a CSMA system, this is the delay waiting for a clear channel to transmit.
Transmit Delay	The time for a radio device to transmit a message over a radio link.
Radio Propagation Delay	The time for a message to propagate through the air to a receiver.
Receiver Processing Delay	The time spent on the receiver side to pass the received message from the device buffer to the application module.

We can abstract from the radio propagation delay, since the transmission range in MANETs typically ranges from 10 to 100 meters, while the radio propagation speed is approximately 3×10^8 meters per second. The media access delay depends on the MAC overhead, such as collisions and waiting time.

4 Formalizing MANET Mobility and Communication

This section presents a modeling framework for MANETs with nodes that communicate wirelessly. Section 4.2 shows how mobile nodes can be specified in Real-Time Maude, Section 4.3 explains how the timed behavior of MANETs can be defined in a way that allows us to easily compose our model with MANET protocols, and Section 4.4 formalizes wireless communication for MANETs.

4.1 Some Basic Data Types

We assume a sort `Location` for the set of locations, a sort `Speed` for the different velocities with which a node can move, a set `Direction` for the different direc-

tions that a node can choose, and sorts `SpeedRange`, `DirRange`, and `DestRange` denoting *sets* of, respectively, `Speed`, `Direction`, and `Location` elements.

We also assume that nodes move in a two-dimensional square with length `areaSize`. A location is therefore represented as a pair x `.` y of rational numbers:[2]

```
op _._ : Rat Rat ~> Location [ctor] .
cmb X . Y : Location if 0 <= X and X <= areaSize /\ 0 <= Y and Y <= areaSize .
```

We do not further specify the different powersets, whose elements could be unions of dense intervals or of single points, or both. Since the nodes need to nondeterministically select a new speed, a new next destination, and/or a new next direction, we assume for generality's sake that there is an operator `choose` that can select any value in the respective set nondeterministically, and an operator `[_]`, so that an element e can be chosen from a set S if and only if there is a rewrite (in zero or more steps) `choose(`S`) => [`e`]`. For example, if we have a discrete set of possible next directions d_1 `;` d_2 `; ...;` d_n, where the set union operator `_;_` is declared to be associative and commutative, we can specify that any value from the set can be selected, by giving the following rewrite rule:

```
var D : Direction .   var DR : DirRange .
rl [chooseDir] :  choose(D ; DR) => [D] .
```

4.2 Modeling Mobile Nodes

We model a MANET node in an object-oriented style, where a mobile node is modeled as an object instance of some subclass of the following base class `Node`:

```
class Node | currentLocation : Location .
```

The attribute `currentLocation` denotes the node's current location. A *stationary* node is an object instance of the subclass `StationaryNode` that does not add any attribute to `Node`:

```
class StationaryNode .          subclass StationaryNode < Node .
```

A mobile node is modeled as an object of a subclass of the class `MobileNode`:

```
class MobileNode | speed : Speed,  direction : Direction,  timer : TimeInf .
subclass MobileNode < Node .
```

where `speed` and `direction` denote, respectively, the node's current speed and its current movement direction. The `timer` attribute is used to ensure that a node changes its movement (or lack thereof) in a timely manner; that is, `timer` denotes the time remaining until some discrete event must take place.

[2] We do not show most variable declarations, but follow the Maude convention that variables are written in capital letters.

Random Walk. A node moving according to the random walk model is continuously moving, in time intervals of length `movingTime`. At the end of an interval, the node nondeterministically chooses a new speed and a new direction for its next interval. Such a node is modeled by an object of the subclass `RWNode`:

```
class RWNode | speedRange : SpeedRange, dirRange : DirRange .
subclass RWNode < MobileNode .
```

where `speedRange` and `dirRange` denote the set of possible next speeds and directions, respectively. The `timer` attribute inherited from its superclass denotes the remaining time of its current move interval. The instantaneous behavior of the mobility part of such a node can be modeled by the following rule. In this rule, the node is finishing one interval (the `timer` attribute is 0), and must select new speed and direction for its next round, and reset the timer:

```
crl [startNewMove] :
   < O : RWNode | timer : 0, speedRange : SR,  dirRange : DR >
 =>
   < O : RWNode | timer : movingTime, speed : S, direction : D > .
  if choose(SR) => [S]  /\ choose(DR) => [D] .
```

The actual movement of such a node is modeled in Section 4.3.

Random Waypoint. In the random waypoint mobility model, a node alternates between pausing and moving. When it starts moving, it selects a new speed and a new destination and starts moving towards the destination. Such a node should be modeled by an object instance of the `RWPNode` subclass:

```
class RWPNode | speedRange : SpeedRange,   destRange : DestRange,
                status : Status .
subclass RWPNode < MobileNode .
```

The `status` attribute is either `pausing` or `moving`, and `destRange` denotes the possible next destinations.

The instantaneous behavior of this mobility model is given by the following rewrite rules. First, if the node is `pausing` and the `timer` expires, the node must get moving by selecting a new speed and desired next location, and resetting the timer so that it expires when the goal location is reached:

```
var MOVE-TIME : Time .

crl [startMoving] :
    < O : RWPNode | currentLocation : CURR-LOC, status : pausing,
                    timer : 0, speedRange : SR,  destRange : DER >
   =>
    < O : RWPNode | status : moving, speed : S,
                    direction : D,  timer : MOVE-TIME >
  if choose(SR) => [S]  /\  choose(DER) => [NEXT-LOC]
     /\ D := direction(L, NEXT-LOC)
     /\ MOVE-TIME := timeBetweenLocations(CURR-LOC, NEXT-LOC, S) .
```

where `direction` gives the direction from one location to another, and `timeBetweenLocations` denotes the time it takes to travel between two locations at a given speed. The selected speed cannot be zero, unless the selected next location is also the current location, because then the last matching equation would not hold, since the traveling time between the two locations would be the infinity value `INF`, which is not a `Time` value.

The following rule applies when the timer of a *moving* node expires; then it is time to take a rest for `pauseTime` time units:

```
rl [startPausing] :
   < O : RWPNode | status : moving, timer : 0 >
  =>
   < O : RWPNode | status : pausing, timer : pauseTime, speed : 0 > .
```

Random direction nodes can be defined in the same way; see [14] for details.

4.3 Timed Behavior and Compositionality

Our model of mobile nodes must be easily *composable* with "application" protocols such as AODV to define a particular MANET system. The straightforward way of composing our model of mobility with a MANET protocol is to let the nodes in the application protocol be modeled as objects of subclasses of the classes introduced above, since a subclass "inherits" all the attributes and rewrite rules of its superclasses; in particular, such application-specific subclasses would inherit the rewrite rules modeling the movements of their nodes.

However, we must allow the user to define the *timed behavior* of her system, and compose it with the timed behavior of mobile nodes. We therefore use the following extension of the "standard" tick rule for object-oriented specifications:

```
var T : Time .    var C : Configuration .
crl [tick] : {C} => {timeEffect(timeEffectMob(C, T), T)} in time T
             if T <= min(mte(C), mteMob(C)) .
```

where `timeEffectMob` defines the effect of time elapse on the mobility-specific parts of the system, and `timeEffect` defines how the passage of time changes the state in the other parts of the composed system. Likewise, `mteMob` denotes the maximum amount of time that may elapse from a given state until some mobility action must be taken, and `mte` defines the amount of time until the application protocol must perform a discrete action. These functions distribute over the objects and messages in the configuration as explained in [14].

Since the speed is 0 when a node is pausing, we can easily define the timed behavior of both stationary and mobile nodes. First of all, time does not affect (the mobility-specific parts of) a stationary node:

```
eq timeEffectMob(< O : StationaryNode | >, T)  =  < O : StationaryNode | >.
```

Time affects a mobile node by moving the node and decreasing its timer value:

```
eq timeEffectMob(< O : MobileNode | currentLocation : L, speed : S,
                                    direction : D, timer : T1 >, T)
 = < O : MobileNode | currentLocation : move(L,S,D,T),
                      timer : T1 monus T > .
```

where `move(`l`,`s`,`d`,`t`)` denotes the location resulting from moving a node in location l for t time units in direction d and with speed s. This function also makes sure that a node does not move beyond the area under consideration.

The mobility model does not restrict the time advance for stationary nodes, whereas for mobile nodes, time can advance until the `timer` becomes 0:

```
eq mteMob(< O : StationaryNode | >) = INF .
eq mteMob(< O : MobileNode | timer : T >) = T .
```

4.4 Modeling Wireless Communication in Mobile Systems

Only nodes that are sufficiently close to the sender, i.e., within the sender's *transmission range*, receive a message with sufficient signal strength. However, both the sender and the potential receivers might move (possibly out of, or into, the sender's transmission range) *during* the entire communication delay.

As mentioned in Section 3, the total communication "delay" can be decomposed into five parts. If we abstract from the radio propagation delay, the per-hop delay can be seen to consist of two parts: the delay at the sender side (including sender processing delay, media access delay, and transmit delay) and the delay at the receiver side (including receiver processing delay). The point is that exactly those nodes that are within the transmission range of the sender *when the sending delay ends* should receive a message.

It is also worth mentioning that our model is still somewhat abstract and does not capture all network factors, most notably collisions.

In MANETs communication can be by broadcast, unicast, or groupcast, depending on which kind of message a transmitter intends to send, and who are the recipients. In our model we have three corresponding message constructors for broadcast, unicast, and groupcast, respectively:

```
msg broadcast_from_ : MsgContent Oid -> Msg .
msg unicast_from_to_ : MsgContent Oid Oid -> Msg .
msg gpcast_from_to_ : MsgContent Oid NeighborSet -> Msg .
```

When a node *sender* wants to broadcast some message content *mc*, it generates a "message" `broadcast` *mc* `from` *sender*. The following equation adds the delay on the sending side, `sendDelay`, to this "broadcast message:"

```
eq broadcast MC from O = dly(transmit MC from O, sendDelay) .
```

The crucial moment is when the sending delay expires and the `transmit` message becomes "ripe." All the nodes that are within the transmission range of the sender *at that moment* should receive the message. This distribution is

performed by the function `distrMsg`, where `distrMsg(`*snd*, *loc*, *mc*, *conf*`)` generates a *single* message, with content *mc*, to each node in *conf* that is currently within the transmission range of location *loc*; furthermore, this single message has delay `recDelay`, modeling the delay at the receiving site:

```
eq {< O : Node | currentLocation : L > (transmit MC from O) C}
 = {< O : Node | > distrMsg(O, L, MC, C)} .

eq distrMsg(O, L, MC, < O' : Node | currentLocation : L' > C)
 = < O' : Node | currentLocation : L' > distrMsg(O, L, MC, C)
   (if L withinTransRangeOf L' then dly((MC from O to O'), recDelay)
    else none fi) .
```

Unicast and groupcast are modeled similarly.

5 Case Study: Route Discovery in AODV

This section first gives an overview of the AODV routing protocol, and then presents our Real-Time Maude model of AODV, focusing on the route discovery process. The entire executable Real-Time Maude specification is available at `http://www.ifi.uio.no/RealTimeMaude/MANET/wrla2014-manets.rtmaude`.

5.1 Route Discovery in AODV

AODV [22] is a widely used algorithm for routing messages between mobile nodes which dynamically form an ad hoc network. AODV allows a source node to initiate a route discovery process on an on-demand basis to establish a route to a destination node.

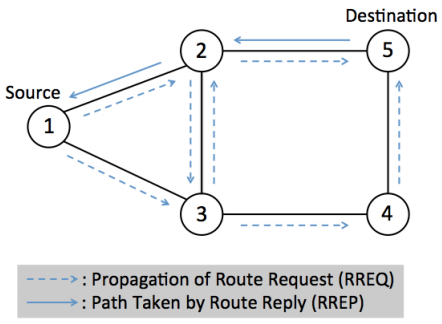


Fig. 2. Route discovery process.

A source node `S` initiates a route discovery process by broadcasting a route request (RREQ) message to its neighbors. An intermediate node can either unicast a route reply (RREP) message back to the source if a valid route to the destination `D` can be found in its local routing table, or rebroadcast the received RREQ to its own neighbors. As the RREQ travels from `S` to `D`, reverse paths from all nodes back to `S` are automatically set up. Eventually, when the RREQ reaches `D`, it sends a RREP back along the previously established reverse path. After this process, a route between `S` and `D` is set up.

5.2 Modeling Route Discovery in Real-Time Maude

Modeling Nodes and Messages. We model an AODV node as an object of a subclass `AODVNode` of class `Node`. The new attributes show the identification of a node's routing request, the sequence number of a node itself, the local routing table, and the buffered routing requests sent since the beginning of the current round, respectively.

```
class AODVNode | rreqID : Nat, sequenceNumber : Sqn,
                 routingTable : RouteTable, requestBuffer : RreqBuffer .
subclass AODVNode < Node .
```

A routing table of sort `RouteTable` is modeled using the predefined data type `MAP`. It consists of routing table entries of the form `Oid |-> Tuple3`, mapping a destination node `Oid` to a 3-element tuple: the next hop towards the destination, the distance to the destination, and the local destination sequence number. A route request buffer of the sort `RreqBuffer` is specified as a set of requests, each of which is of the form `Oid` $\sim$ `Sqn`, and uniquely identifies a route request by the identifier of a node and its sequence number.

In the AODV route discovery process there are mainly two kinds of messages: RREQ and RREP. They are specified in our model as `rreq(...)` and `rrep(...)` respectively. The message content will be illustrated below.

Modeling Route Discovery. A route discovery process in AODV consists of three parts: initiating route discovery, route request handling, and route reply handling. We only illustrate part of the route request handling, and refer the reader to our longer report [14] for more details.

The RREQ-handling rules specify all events that may happen when a route request is received. The receiving node first checks whether a received `(OIP` $\sim$ `RREQID)` has already been stored locally in the request buffer. If so, the route request is ignored and the local routing table is updated by adding a routing table entry towards the sender; otherwise, the receiving node adds the new route request identifier to the request buffer, and takes further actions according to the roles played by the receiving node. In the following case, the receiving node is an intermediate node.

When receiving the RREQ message, an intermediate node either: (a) generates a route reply to the sender, or (b) re-broadcasts the received RREQ to its neighbors. For example, action (a), as the following rewrite rule shows, happens only when `O`'s local information is fresher than that in the RREQ message (`DSN <= localdsn(RT[DIP])`). Then `O` unicasts the route reply with the fresher destination sequence number and its distance in hops from the destination along the route back to the source node.

```
crl [on-receiving-rreq-3] :
    (rreq(OIP,OSN,RREQID,DIP,DSN,HOPS,SIP) from SIP to O)
    < O : AODVNode | routingTable : RT, requestBuffer : RB >
```

```
 => < O : AODVNode | routingTable : RT'',
                     requestBuffer : (OIP ~ RREQID, RB) >
    (msg rrep(OIP,DIP,localdsn(RT''[DIP]),hops(RT''[DIP]),O)
       from O to nexthop(RT''[OIP]))
    if RT' := update(SIP,SIP,1,O,RT) /\
       RT'' := update(OIP,SIP,HOPS + 1,OSN,RT') /\
       not (OIP ~ RREQID) in RB /\ inRT(RT,DIP) /\
       DIP =/= O /\ DSN <= localdsn(RT[DIP]) .
```

6 Formal Analysis of AODV

In this section we analyze the AODV route discovery process under different mobility models. We therefore define node objects that belong to a subclass of both `AODVNode` and a class defining the desired mobility pattern. For example, a node moving according to the random waypoint model is an object instance of the class `RWPANode`, and a stationary node is an instance of the class `SANode`:

```
class RWPANode .        subclass RWPANode < RWPNode AODVNode .
class SANode .          subclass SANode < StationaryNode AODVNode .
```

The main objective of a routing protocol such as AODV is that a route between the desired source and the desired destination is eventually established. To analyze this property, we define a parameterized atomic proposition `route-found(SRC,DEST)` to hold if we can find, in the routing table of the source node `SRC`, a routing table entry towards the destination node `DEST`:

```
op route-found : Oid Oid -> Prop [ctor] .
eq {< SRC : AODVNode | routingTable : RT , (DEST |-> TP) >  REST}
      |= route-found(SRC, DEST) = true .
```

The desired property of AODV can then be formalized as the temporal logic formula `<> route-found(...)`. Given an initial state `initConfig`, the following command returns `true` if our desired property holds in the first test round (`roundTime`); otherwise, a trace showing a counterexample is provided.

```
(mc {initConfig} |=t <> route-found(src,dest) in time <= roundTime .)
```

Experiment Scenarios. We define the following setting for our experiments:

- The transmission range is `10m`, and the test area is `100m` × `100m`.
- The test round is `100s`. The delay at the sender and at the receiver is set to `10s` and `5s`, respectively.
- The range of possible velocities is the singleton set `(1)`.
- Nodes can move right, up, left or down: the direction range is a subset of `(0,90,180,270)`, and the destination range is a subset of four locations in the corresponding four directions based on a node's current location.

We have analyzed AODV in seven different scenarios; five of them are described below and the other cases are described in our longer report [14]:

- *Scenario (i)*, shown in Fig. 2, has five *stationary* nodes, where node 1, located at (45 . 45), wants to build a route to node 5, located at (60 . 50), and nodes 2, 3 and 4 are at (50 . 50), (50 . 40), and (60 . 40), respectively.
- *Scenario (i')*, shown in Fig. 3 (a solid circle refers to the initial location of a node, while a dashed circle refers to some point along the motion path of a node), has the same topology as Scenario (i), but now node 2 is a *random waypoint* node that can move up. We set its pause time to: (a) 10s, (b) 30s, or (c) 60s. The initial state of this scenario is specified as:

```
eq src = 1 .
eq initConfig =
    (bootstrap src)
    < 1 : SANode | currentLocation : 45 . 45 , rreqID : 10, sequenceNumber : 1,
                   routingTable : empty,  requestBuffer : empty >
    < 2 : RWPANode | currentLocation : 50 . 50, speed : 0, direction : 0, timer : pauseTime,
                     speedRange : (1), destRange : (50 . 60), status : pausing, rreqID : 20,
                     sequenceNumber : 1, routingTable : empty, requestBuffer : empty >
    < 3 : SANode | currentLocation : 50 . 40, ... >
    < 4 : SANode | currentLocation : 60 . 40, ... >
    < 5 : SANode | currentLocation : 60 . 50, ... >  .
```

- *Scenario (ii),* also shown in Fig. 3, has three nodes with both nodes 2, located at (40 . 50), and 3 (a random waypoint node located at (50 . 40)) intending to build a route to the destination node 1 located at (50 . 50).

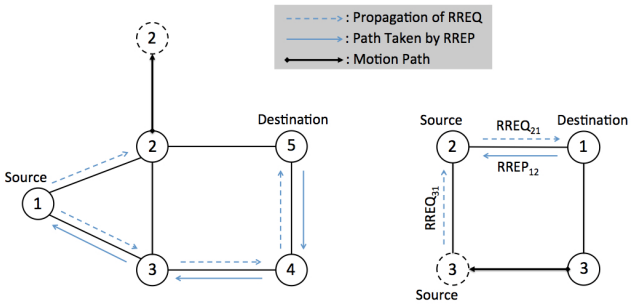


Fig. 3. Scenarios (i') and (ii)

Analysis Results. The results of the model checking show that the desired property holds in Scenarios (i), (i')-(a), and (i')-(c), but not in Scenarios (i')-(b) and (ii).

In Scenario (i')-(b), the pause time (30s) allows node 2 to forward the RREQ message from node 1 to node 5. However, node 2 cannot receive the RREP message from node 5 due to its movement (the dash circle in this case is at (50 . 60)). Meanwhile, since node 5 has already recorded node 1's RREQ from node 2, it ignores the one from node 4.

In Scenario (ii), before sending out the RREQ message, node 3 moves left to a new location (40 . 40) within the transmission range of node 2. Thus, to establish the route to node 1, node 3's RREQ message needs to be forwarded by node 2. However, the model checking counterexample shows that route discovery for node 3 fails: no route can be found between nodes 3 and 1, though obviously node 2 succeeds in building a route to node 1. This problem arises

due to the discarding of the RREP message. As stated in [22], an intermediate node forwards a RREP message only if the RREP message serves to update its routing table entry towards the destination. However, in this case, node `2` has already secured an optimal route to node `1` before receiving the RREQ message from node `3`. [7] also pointed out this problem, but in a static linear topology with three nodes.

7 Related Work

There are a number of formal specification and analysis efforts of MANETs in general, and AODV in particular.

Bhargavan et al. [1] use the SPIN model checker to analyze AODV. They only consider a 3-node topology with one link break, but without node movement, and communication delay is not considered. Chiyangwa et al. [4] apply the real-time model checker Uppaal to analyze AODV. They only consider a static linear network topology. Although they take communication delay into account, the effect of mobility on communication delay is not considered, since the topology is fixed. Fehnker et al. [7] also use Uppaal to analyze AODV. They also only considered static topologies, or simple dynamic topologies by adding or removing a link, and those topologies are based on the connectivity graph without concrete locations for nodes. Furthermore, no timing issues are considered. Höfner et al. [12] apply statistical model checking to AODV. However, mobility is simply considered by arbitrary instantaneous node jumping between zones that split the whole test grid. Although they take into account the communication delay, the combination of mobility and communication delay is not considered. None of these studies has built a generic framework for MANETs. Our modeling framework aims at the combination of wireless communication and mobility, and allows formal modeling and analysis of protocols under realistic mobility models.

On the process algebra side, CWS [17], CBS# [19], CMAN [10], CMN [15], the ω-calculus [23], RBPT [9], [11], TCWS [16] and AWN [6], have been proposed as process algebraic modeling languages for MANETs. These languages feature a form of local broadcast, in which a message sent by a node could be received by other nodes "within transmission range." However, the connectivity is only considered abstractly and logically, without taking into account concrete locations and transmission range for nodes. Furthermore, [17] only considers fixed network topologies, whereas the others (except [11]) deal with arbitrary changes in topology. Godskesen et al. [11] consider realistic mobility, and propose concrete mobility models. However, no protocol application or automated analysis is given, and communication delay is not taken into account. Merro et al. [16] propose a timed calculus with time-consuming communications, and equip it with a formal semantics to analyze communication collisions.

Generally, these studies have proposed a framework for MANETs, but they lack of either mobility modeling or timing issues handling.

There are also a number of well known "ambient" calculi for mobility, such as the ambient calculus [3], the π-calculus [18], and the join-calculus [8]. However,

these are very abstract models that do not take locations and geographically bounded communication into account, and are therefore not suitable to model MANETs at the level of abstraction considered in this paper.

Finally, Maude and Real-Time Maude have been applied to analyze wireless sensor networks, but the work in [21,13] do not consider node mobility (even though [13] mentions that mobility is addressed in a technical report in preparation; however, we cannot find that technical report).

8 Concluding Remarks

We have defined in Real-Time Maude what we believe is the first formal model of MANETs that provides a reasonably faithful model of popular node movement patterns and wireless communication. We have used our compositional model to specify and formally analyze the AODV routing protocol, and have shown that such Real-Time Maude analysis could easily find the known flaw in AODV.

We have abstracted from message collision, which should also be considered in our model. The price to pay for having a much more realistic model of MANETs than other formal approaches is that the state space quickly becomes too large for model checking. We should therefore develop statistical model checking techniques for MANETs. Most importantly, we should develop abstraction techniques for MANETs. The formalization presented in this paper has provided the necessary foundation for such efforts.

Acknowledgments. We thank the anonymous reviewers for helpful comments on a previous version of this paper. This work has been partially supported by AFOSR Contract FA8750-11-2-0084 and NSF Grant CNS 13-19109.

References

1. Bhargavan, K., Obradovic, D., Gunter, C.: Formal verification of standards for distance vector routing protocols. Journal of the ACM 49(4), 538–576 (2002)
2. Camp, T., Boleng, J., Davies, V.: A survey of mobility models for ad hoc network research. Wireless Communications and Mobile Computing 2(5), 483–502 (2002)
3. Cardelli, L., Gordon, A.D.: Mobile ambients. In: Proc. POPL'98. ACM (1998)
4. Chiyangwa, S., Kwiatkowska, M.Z.: A timing analysis of AODV. In: Proc. FMOODS'05. LNCS, vol. 3535. Springer (2005)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
6. Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.: A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. In: Tech. Rep. 5513. NICTA (2012)
7. Fehnker, A., van Glabbeek, R.J., Hfner, P., McIver, A., Portmann, M., Tan, W.L.: Automated analysis of AODV using Uppaal. In: Proc. TACAS'12. LNCS, vol. 7214. Springer (2012)
8. Fournet, C., Gonthier, G.: The reflexive cham and the join-calculus. In: Proc. POPL'96. ACM (1996)

9. Ghassemi, F., Fokkink, W., Movaghar, A.: Restricted broadcast process theory. In: Proc. SEFM '08. IEEE (2008)
10. Godskesen, J.C.: A calculus for mobile ad hoc networks. In: Proc. Coordination'07. LNCS, vol. 4467. Springer (2007)
11. Godskesen, J.C., Nanz, S.: Mobility models and behavioural equivalence for wireless networks. In: Proc. Coordination'09. LNCS, vol. 5521. Springer (2009)
12. Höfner, P., Kamali, M.: Quantitative analysis of AODV and its variants on dynamic topologies using statistical model checking. In: Proc. FORMATS'13. LNCS, vol. 8053. Springer (2013)
13. Katelman, M., Meseguer, J., Hou, J.C.: Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking. In: Proc. FMOODS'08. LNCS, vol. 5051. Springer (2008)
14. Liu, S., Ölveczky, P., Meseguer, J.: A framework for mobile ad hoc networks in Real-Time Maude (2014), `http://www.ifi.uio.no/RealTimeMaude/MANET/wrla14-manets-tech.pdf`
15. Merro, M.: An observational theory for mobile ad hoc networks (full version). Inf. Comput. 207(2), 194–208 (2009)
16. Merro, M., Ballardin, F., Sibilio, E.: A timed calculus for wireless systems. Theor. Comput. Sci. 412(47), 6585–6611 (2011)
17. Mezzetti, N., Sangiorgi, D.: Towards a calculus for wireless systems. Electron. Notes Theor. Comput. Sci. 158, 331–353 (2006)
18. Milner, R.: Communicating and mobile systems – the Pi-calculus. Cambridge University Press (1999)
19. Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. Theor. Comput. Sci. 367(1), 203–227 (2006)
20. Ölveczky, P., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-order and Symbolic Computation 20(1-2), 161–196 (2007)
21. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theoretical Computer Science 410(2-3), 254–280 (2009)
22. Perking, C., Belding-Royer, E., Das, S.: Ad hoc on-demand distance vector (AODV) routing. RFC 3561 (experimental) (2003), `http://www.ietf.org/rfc/rfc3561`
23. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. Sci. Comput. Program. 75(6), 440–469 (2010)
24. Su, P.: Delay measurement time synchronization for wireless sensor networks. Intel Research Berkeley Lab (2003)

Strong and Weak Operational Termination of Order-Sorted Rewrite Theories*

Salvador Lucas[1,2] and José Meseguer[2]

[1] DSIC, Universitat Politècnica de València, Spain
[2] CS Dept. University of Illinois at Urbana-Champaign, IL, USA

Abstract. This paper presents several new results on conditional term rewriting within the general framework of order-sorted rewrite theories (OSRTs) which contains the more restricted framework of conditional term rewriting systems (CTRSs) as a special case. The results uncover some subtle issues about conditional termination. We first of all generalize a previous known result characterizing the operational termination of a CTRS by the quasi-decreasing ordering notion to a similar result for OSRTs. Second, we point out that the notions of *irreducible* term and of *normal form*, which coincide for unsorted rewriting are *totally different* for conditional rewriting and formally characterize that difference. We then define the notion of a *weakly operationally terminating* (or *weakly normalizing*) OSRT, give several evaluation mechanisms to compute normal forms in such theories, and investigate general conditions under which the rewriting-based operational semantics and the initial algebra semantics of a confluent OSRT coincide thanks to a notion of *canonical term algebra*. Finally, we investigate appropriate conditions and proof methods to ensure good executability properties of an OSRT for computing normal forms.

Keywords: Conditional term rewriting, strong and weak operational termination, irreducible terms, normalized terms, rewriting logic, Maude.

1 Introduction

This paper presents several new contributions to conditional term rewriting and to the semantics of declarative, rewriting-based languages. The key notion is that of an *order-sorted rewrite theory* (OSRT) $\mathcal{R} = (\Sigma, B, R)$, where (Σ, B) is an order-sorted equational theory [9] with equational axioms B, and R is a collection of rewrite rules with oriented conditions of the form: $\ell \to r$ if $s_1 \to t_1, \ldots, s_n \to t_n$, which are applied *modulo* B. All the results are *in particular* new results for *contitional term rewriting systems* (CTRSs); that is, for order-sorted

* Research partially supported by NSF grant CNS 13-19109. Salvador Lucas' research was developed during a sabbatical year at the CS Department of the UIUC and was also partially supported by Spanish MECD grant PRX12/00214, MINECO project TIN2010-21062-C02-02, and GV project PROMETEO/2011/052.

rewrite theories of the form $\mathcal{R} = (\Sigma, \varnothing, R)$, with Σ having a *single* sort. The greater generality of OSRTs is not a caprice, but an absolute necessity for making formal specification and declarative programming practical and expressive.

Our contributions consist in asking and providing detailed answers to the following, innocent-sounding questions:

1. Can the operational termination of OSRTs be characterized in terms of orders?
2. What is the right notion of *normal form* for an OSRT?
3. What is the right notion of *weak operational termination* for an OSRT?
4. Under what conditions can OSRTs be used as *declarative programs* having a well-behaved semantics? And how can we *evaluate* such programs?
5. Under what conditions does an OSRT have a *canonical term algebra* that can be effectively computed and that provides a *complete agreement* between the operational semantics of the OSRT as a functional program, and its mathematical, initial algebra semantics? How can some of these executability conditions be checked in practice?

Surprisingly enough, some of these questions seem to never have been asked. At best, the issues involved seem to have remained implicit as not well-understood, anomalous features in the literature. Consider, for example, question (2) above, which asks about the notion of normal form. For unconditional term rewriting the notion is absolutely clear and unproblematic: a normal form is a term t that is *irreducible*, that is, such that there is no t' with $t \to t'$. For an OSRT, and in particular for a CTRS, the notion of normal form is actually highly problematic. The big problem is that for an OSRT there can be terms t that are irreducible in the above sense, i.e., there is no t' with $t \to t'$, but such that when we give t to a rewrite engine for evaluation such an engine loops! For a trivial example, consider the single conditional rewrite rule $a \to b$ if $a \to c$. Since the rewrite relation defined by this conditional rule is the empty set, the constant a is trivially irreducible; but the proof tree associated to the normalization of a using the CTRS inference system is *infinite* [7], and a rewrite engine that tries to evaluate a will loop when trying to satisfy the rule's condition.[3] Therefore, calling a a *normal form* is a very bad joke, since, intuitively, a term is considered to be a normal form if it is "fully normalized," that is, if it is the *result* of fully evaluating some input term by rewriting. Our answer to this puzzle is to introduce a precise distinction (fully articulated in the paper) between irreducible terms and normal forms: every normal form is irreducible, but, as the above example shows, not every irreducible term is a normal form. We call an OSRT *normal* iff every irreducible term is a normal form, and call it *abnormal* otherwise. Abnormal theories, like the one above, are hopeless for executablity purposes and should be viewed as monsters in the menagerie of OSRTs.

Termination is quite a subtle issue for OSRTs in general and CTRSs in particular. Many notions have been proposed (see e.g., [11]), but it is by now

[3] For this trivial example one could find ways for an engine to detect *this* looping; but undecidability of termination makes a general loop-detecting engine an oxymoron.

well-understood that the most satisfactory notion from a computational point of view is that of *operational termination* [7] (more on this later). Here we ask and answer two questions, further developing this notion. The first is question (1) above. For the case of deterministic 3-CTRS we proved in [7] that operational termination is *equivalent* to the order-based notion of *quasi-decreasingness*. In Section 3 we generalize this result to a similar result characterizing operational termination of OSRTs in terms of an (axiom-compatible) term ordering.

A second, related question, seemingly not previously addressed in the literature, is question (3), which could be rephrased as follows: what is the right notion of weak termination/normalization for OSRTs? As further explained in Section 4, there are in fact two notions, a computationally ill-behaved one (*weak termination*: every term has a terminating rewrite sequence ending in an irreducible term), and a computationally well-behaved one (*weak operational termination*: every term has a normal form).

The notions of normal OSRT and of weak operational termination are closely related to another question, namely, question (4), on executability conditions for declarative, conditional rule-based programs, and on their evaluation methods, i.e., their operational semantics. Interestingly enough, as we explain in Section 4, there are *several* evaluation methods, which become more and more efficient as we impose further conditions on the OSTR which we use as our program.

For *functional* programs specified by an OSRT, the issue is not just one of having good executability conditions, but actually of *correctness*. More precisely, of *semantic agreement* between an abstract *initial algebra semantics* when the rules are viewed as equations, and an *operational semantics* based on rewriting, where the computed *values* —that is, the normal forms— give rise to a very intuitive algebra, the *canonical term algebra*, which under the assumptions of confluence, coherence, sort-decreasingness and operational termination is *isomorphic* to the initial algebra of the specification. Question (5) above asks, essentially: what is the *non plus ultra* in terms of generality to maintain this isomorphism and keeping an exact agreement between mathematical and operational semantics? That is, what are the right conditions for this semantic agreement when we drop the operational termination condition? This is also answered in Section 4, relating the answers to associated evaluation methods to compute normal forms. Last but not least, in Sections 4 and 5 we investigate appropriate *conditions* and *proof methods* to ensure that a theory has good executability properties such as being normal, and evaluation to normal form defining a total recursive function.

2 Preliminaries

Order-Sorted Algebra. We summarize here material from [4, 9] on order-sorted algebra. We start with a partially ordered set $(S, \leq)$ of *sorts*, where $s \leq s'$ is interpreted as *subsort inclusion*. The *connected components* of $(S, \leq)$ are the equivalence classes $[s]$ corresponding to the least equivalence relation $\equiv_{\leq}$ containing $\leq$. We also define $\lfloor s \rfloor = \{s' \in S \mid s' \leq s\}$, i.e., the sorts in S which are smaller than or equal to s. When $[s]$ has an upper bound, we denote it by

$\top_{[s]}$. An order-sorted signature $(\Sigma, S, \leq)$ consists of a poset of sorts $(S, \leq)$ and a $S^* \times S$-indexed family of sets $\Sigma = \{\Sigma_{w,s}\}_{(w,s)\in S^*\times S}$, which are *function symbols* with a given string of argument sorts and a result sort. If $f \in \Sigma_{s_1\ldots s_n,s}$, then we display the function symbol f as $f : s_1 \ldots s_n \longrightarrow s$. This is called a *rank* declaration for symbol f. Some of these symbols f can be *subsort-overloaded*, i.e., they can have several rank declarations related in the $\leq$ ordering [4]. Constant symbols, however, have only one rank declaration. To avoid ambiguous terms, we assume that Σ is *sensible*, meaning that if $f : s_1 \cdots s_n \to s$ and $f : s'_1 \cdots s'_n \to s'$ are such that $[s_i] = [s'_i]$, $1 \leq i \leq n$, then $[s] = [s']$. Throughout this paper, Σ will always be assumed *sensible*.

Given an S-sorted set $\mathcal{X} = \{\mathcal{X}_s \mid s \in S\}$ of *mutually disjoint* sets of variables, the set $\mathcal{T}(\Sigma, \mathcal{X})_s$ of terms of sort s is the least set such that $\mathcal{X}_s \subseteq \mathcal{T}(\Sigma, \mathcal{X})_s$. We let $\mathcal{T}(\Sigma, \mathcal{X})_{[s]} = \bigcup_{s'\in[s]} \mathcal{T}(\Sigma, \mathcal{X})_{s'}$. If $s' \leq s$, then $\mathcal{T}(\Sigma, \mathcal{X})_{s'} \subseteq \mathcal{T}(\Sigma, \mathcal{X})_s$; and if $f : s_1 \ldots s_n \longrightarrow s$ is a rank declaration for symbol f and $t_i \in \mathcal{T}(\Sigma, \mathcal{X})_{s_i}$ for $1 \leq i \leq n$, then $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{X})_s$. The assumption that Σ is sensible ensures that if $[s] \neq [s']$, then $\mathcal{T}(\Sigma, \mathcal{X})_{[s]} \cap \mathcal{T}(\Sigma, \mathcal{X})_{[s']} = \varnothing$.

The set $\mathcal{T}(\Sigma, \mathcal{X})$ of order-sorted terms is $\mathcal{T}(\Sigma, \mathcal{X}) = \cup_{s\in S}\mathcal{T}(\Sigma, \mathcal{X})_s$. An element of any set $\mathcal{T}(\Sigma, \mathcal{X})_s$ is called a *well-formed* term. A simple syntactic condition on $(\Sigma, S, \leq)$ called *preregularity* [4] ensures that each well-formed term t has always a *least sort* possible among all sorts in S, which is denoted $LS(t)$. An order-sorted substitution σ is an S-sorted mapping $\sigma = \{\sigma : \mathcal{X}_s \to \mathcal{T}(\Sigma, \mathcal{X})_s\}_{s\in S}$ from variables to terms. The application of an OS-substitution σ to t (denoted $\sigma(t)$) consists of simultaneously replacing the variables occurring in t by a term according to the mapping σ. A specialization ν is an injective OS-substitution that maps a variable x of sort s to a variable x' of sort $s' \leq s$.

Order-Sorted Rewrite Theories. An (order-sorted) rewrite rule is an ordered pair (l, r), written $l \to r$, with $l, r \in \mathcal{T}(\Sigma, \mathcal{X})$, and $LS(l) \equiv_{\leq} LS(r)$. An *order-sorted conditional rewrite theory* (OSRT) is a triple $\mathcal{R} = (\Sigma, B, R)$, where Σ is an order-sorted signature, B is a set of Σ-equations, and R is a collection of conditional rewrite rules with oriented conditions of the form $\ell \to r$ if $s_1 \to t_1, \ldots, s_n \to t_n$, where $\ell \to r$ and the $s_i \to t_i$ are order-sorted rewrite rules (with $\ell \notin \mathcal{X}_s$ for all $s \in S$), and where the conditions $s_i \to t_i$ are intended to express the reachability of (instances of) t_i from (instances of) s_i. Throughout this paper the equations $(u = v) \in B$ are assumed to be: (i) *regular* (i.e., $\mathcal{V}ar(u) = \mathcal{V}ar(v)$), (ii) *linear* (i.e., no repeated variables in either u or v); (iii) there is a B-matching algorithm; and (iv) *sort-preserving* (i.e., for each substitution θ, $LS(\theta(u)) = LS(\theta(v))$). Examples of axioms B satisfying (i)–(iii) include combinations of associativity and/or commutativity and/or identity axioms. Maude supports rewriting modulo such axioms and also checks automatically property (iv) (it actually checks a somewhat weaker condition for identity axioms that still ensures a least sort for each B-equivalence class).

Rewrite rules $\ell \to r$ if c in OSRTs are classified according to the distribution of variables among ℓ, r, and c, as follows: type 1, if $\mathcal{V}ar(r) \cup \mathcal{V}ar(c) \subseteq \mathcal{V}ar(\ell)$; type 2, if $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(\ell)$; type 3, if $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(\ell) \cup \mathcal{V}ar(c)$; and type 4, if no restriction is given. An n-OSRT contains only rewrite rules of types $m \leq n$. A

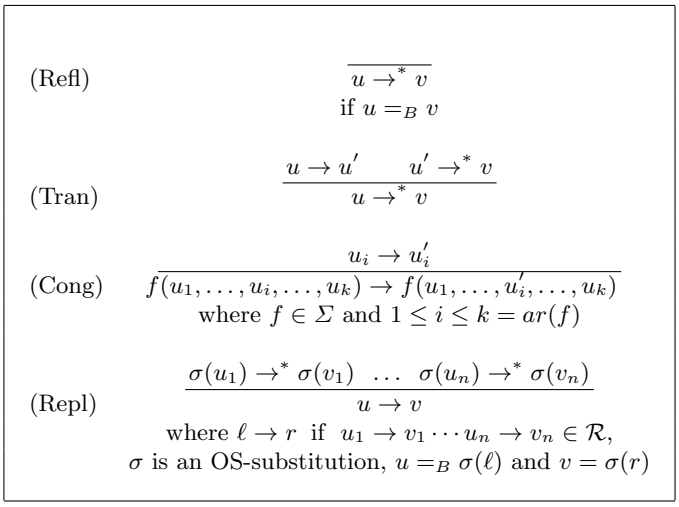


Fig. 1. Inference rules for order-sorted rewrite theories

3-OSRT $\mathcal{R}$ is called *deterministic* if for each rule $l \to r$ if $s_1 \to t_1, \ldots, s_n \to t_n$ in $\mathcal{R}$ and each $1 \leq i \leq n$, we have $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(l) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(t_j)$.

We write $t \to_{\mathcal{R}} u$ (resp. $t \to_{\mathcal{R}}^* u$) iff there is a well-formed proof tree for $t \to u$ (resp. $t \to^* u$) for $\mathcal{R}$ using the inference system in Figure 1. As usual, $\to_{\mathcal{R}}$ is the *one-step* rewrite relation for the OSRT $\mathcal{R}$ and $\to_{\mathcal{R}}^*$ is the *zero-or-more-steps* rewrite relation for $\mathcal{R}$. We write $t \to_{\mathcal{R}}^0 u$ if $t =_B u$; $t \to_{\mathcal{R}}^1 u$ if $t \to_{\mathcal{R}} u$, and $t \to_{\mathcal{R}}^n u$, for some $n > 1$ if there is a term t' such that $t \to_{\mathcal{R}} t'$ and $t' \to_{\mathcal{R}}^{n-1} u$.

Operational Termination. Given a logic $\mathcal{L}$ (defined by its inference rules), one has the notion of a *theory* or *specification* $\mathcal{S}$ in such a logic, so that $\mathcal{L}$'s inference system becomes specialized to each such specification $\mathcal{S}$ to derive its provable theorems φ. Assume that we have an *interpreter* for the logic $\mathcal{L}$, that is, an *inference machine* that, given a theory $\mathcal{S}$ and a goal formula φ will try to incrementally build a proof tree for φ. Intuitively, we will call $\mathcal{S}$ *terminating* if for any φ the interpreter either finds a proof in finite time, or fails in all possible attempts also in finite time. In the same vein, we can say that a predicate π (for instance, $\to$ or $\to^*$ in the inference system of Figure 1) is operationally terminating if for any goal φ such that $\varphi = \pi(t_1, \ldots, t_k)$ for terms $t_1, \ldots, t_k$, φ is operationally terminating. The notion of *operational termination* captures this fact, meaning that, given an initial goal, an interpreter will either succeed in finite time in producing a closed proof tree, or will fail in finite time, not being able to close or extend further any of the possible proof trees, after exhaustively searching all such proof trees [7]. In the following, according to the previous discussion, we speak about operational 1-termination of a OSRT as the operational termination of $\to$ (with respect to the inference system of Figure 1). By operational termination of an OSRT we then mean the operational termination

of $\rightarrow^*$. Similarly, we say that a term t is operationally (1-)terminating if every goal $t \rightarrow^* u$ (resp. $t \rightarrow u$) is operationally terminating for all terms u.

One last issue important for executability purposes is (strong) *B-coherence*. This means that if $t \rightarrow^1_{\mathcal{R}} u$ and $t =_B t'$, then there exists a u' such that $t' \rightarrow^1_{\mathcal{R}} u'$ and $u =_B u'$. For axioms B such as combinations of associativity, commutativity and identity, Maude automatically completes the user-specified rules so that they become B-coherent. In this paper we will assume that *all OSRTs are B-coherent.*

3 Orderings, Quasi-Decreasingness, and (Strong) Operational Termination

A binary relation R on a set A is *terminating* (or well-founded) if there is no infinite sequence $a_1\ R\ a_2\ R\ a_3 \cdots$. Given $f : A^k \rightarrow A$ and $i \in \{1, \ldots, k\}$, we say that f is *i-monotonic* on its i-th argument (or that f is *i-monotone* with respect to R) if $f(x_1, \ldots, x_{i-1}, x, \ldots, x_k)\ R\ f(x_1, \ldots, x_{i-1}, y, \ldots, x_k)$ whenever $x\ R\ y$, for all $x, y, x_1, \ldots, x_k \in A$. We say that R is monotonic if, for all symbols f, f is monotonic w.r.t. R. In [7] we have shown that operational termination of deterministic 3-CTRSs (which are special deterministic 3-OSRTs where the set of sorts S contains a single sort and the set of equations B is empty) is equivalent to *quasi-decreasingness*, i.e., the existence of a well-founded partial ordering $\succ$ on terms satisfying that: (1) the one-step rewriting relation is contained in $\succ$: $\rightarrow_{\mathcal{R}} \subseteq \succ$, (2) the strict subterm relation is contained in $\succ$: $\rhd \subseteq \succ$, and (3) for every rule $\ell \rightarrow r$ if $s_1 \rightarrow t_1, \ldots, s_n \rightarrow t_n$, substitution σ, and index i, $0 \leq i < n$, if $\sigma(s_j) \rightarrow^*_{\mathcal{R}} \sigma(t_j)$ for every $1 \leq j \leq i$, then $\sigma(l) \succ \sigma(s_{i+1})$. In the following, we generalize this result for deterministic 3-OSRTs under the assumptions on B stated in Section 2. We use *strong* operational termination and operational termination as synonymous. This is done to distinguish it from a notion of *weak* operational termination presented later. Now we address the problem of defining appropriate orderings for dealing with order-sorted terms and rewrite theories.

3.1 Orderings for Order-Sorted Terms

A strict ordering $\succ_s$ on terms of sort s is an irreflexive and transitive binary relation on $\mathcal{T}(\Sigma, \mathcal{X})_s$. A strict ordering $\succ_{[s]}$ on terms of sort in the connected component $[s]$ (of $S/\equiv_{\leq}$) is an irreflexive and transitive binary relation on $\mathcal{T}(\Sigma, \mathcal{X})_{[s]}$.

Remark 1. Order-sorted rewriting proceeds by transforming terms of the same connected component $[s] \in S/\equiv_{\leq}$. Therefore, orderings $\succ_{[s]}$ indexed by connected components of sorts, rather than by sorts, are more appropriate for compatibility with the order-sorted rewrite relation. Indeed, note that $\rightarrow^+_{\mathcal{R}} = (\rightarrow^+_{\mathcal{R}[s]})$ is a well-founded S-ordering if the one-step rewrite relation is terminating, an that it is monotonic if $\mathcal{R}$ is sort-decreasing. On the other hand, we can always obtain an ordering $\succ_s$ on terms of sort s as follows: $\succ_s = \succ_{[s]} \cap \mathcal{T}(\Sigma, \mathcal{X})^2_s$.

A strict S-ordering $\succ_S = \{\succ_{[s]}\}_{[s] \in S/\equiv_{\leq}}$ is an S-sorted strict ordering on $\mathcal{T}(\Sigma, \mathcal{X})$, i.e., given terms $u, v \in \mathcal{T}(\Sigma, \mathcal{X})$, $u \succ_S v$ if and only if $u, v \in \mathcal{T}(\mathcal{F}, \mathcal{X})_{[s]}$ for some

$[s] \in S/\equiv_{\leq}$ and $u \succ_{[s]} v$. An S-ordering $\succ_S$ is: *well-founded* if its components $\succ_{[s]}$ are well-founded for all $s \in S$; *stable* if for all S-sorted substitution σ, $s \in S$, and terms $u, v \in \mathcal{T}(\Sigma, \mathcal{X})_{[s]}$ $u \succ_{[s]} v$, then $\sigma(u) \succ_{[s]} \sigma(v)$; *monotonic* if for all $f : s_1 \cdots s_k \to s \in \Sigma$ and terms $u_i, v_i \in \mathcal{T}(\Sigma, \mathcal{X})_{[s_i]}$ for $1 \leq i \leq k$, if $u_i \succ_{[s_i]} v_i$, then $f(u_1, \ldots, u_i, \ldots, u_k) \succ_{[s]} f(u_1, \ldots, u_i, \ldots, u_k)$. An S-ordering $\succ_S$ on $\mathcal{T}(\Sigma, \mathcal{X})$ is compatible with a set of equations B on $\mathcal{T}(\Sigma, \mathcal{X})$ if for all terms u, u', v, whenever $u \succ_S v$ and $u' =_B u$, we have $u' \succ_S v$ (in short: $=_B \circ \succ \subseteq \succ$). The previous definitions generalize to arbitrary relations (quasi-orderings $\gtrsim$, equivalences $\approx$, etc.) on order-sorted terms.

Remark 2. S-sorted orderings cannot compare terms of different connected components. Still, S-sorted orderings are the natural ones when comparing the left- and right-hand sides of the rules of an order-sorted (conditional) rewrite system.

A term ordering $\succ$ is a strict order on $\mathcal{T}(\Sigma, \mathcal{X})$. An S-sorted ordering $\succ_S$ on $\mathcal{T}(\Sigma, \mathcal{X})$ defines a term ordering on $\mathcal{T}(\Sigma, \mathcal{X})$: $u \succ v$ iff $\exists [s] \in S/\equiv_{\leq}$ such that $u \succ_{[s]} v$. A term ordering which is *not* S-sorted is the subterm relation $\trianglerighteq$: $\forall u, v \in \mathcal{T}(\Sigma, \mathcal{X})$, $u \trianglerighteq v$ if either $u = v$ or $u = f(u_1, \ldots, u_k)$ for some $f : s_1 \cdots s_k \to s \in \Sigma$ and $u_i \trianglerighteq t$ for some i, $1 \leq i \leq k$. We write $u \triangleright v$ if $u \trianglerighteq v$ and $u \neq v$.

3.2 Quasi-Decreasingness and (Strong) Operational Termination of determinisitc 3-OSRTs

After the previous discussion, we can provide a generalization to determinisitc 3-OSRTs of the usual notion of quasi-decreasingness for determinsitic 3-CTRSs.

Definition 1 (Quasi-decreasingness). *A deterministic 3-OSRT (Σ, B, R) is* quasi-decreasing *if there is a well-founded term ordering $\succ$ on $\mathcal{T}(\Sigma, \mathcal{X})$ satisfying: (1) $\to_{\mathcal{R}} \subseteq \succ$, (2) $=_B \circ \succ \subseteq \succ$, (3) $\triangleright \subseteq \succ$, and (4) for every rule $l \to r$ if $u_1 \to v_1, \ldots, u_n \to v_n$, S-sorted substitution σ, and index i, $0 \leq i < n$, if $\sigma(u_j) \to_{\mathcal{R}}^* \sigma(v_j)$ for every $1 \leq j \leq i$, then $\sigma(l) \succ \sigma(s_{i+1})$.*

Quasi-decreasingness is a sufficient condition for operational termination of deterministic 3-OSRTs.

Theorem 1. *Let $\mathcal{R}$ be a deterministic 3-OSRT. If $\mathcal{R}$ is quasi-decreasing, then it is operationally terminating.*

Quasi-decreasingness is also necessary for operational termination of order-sorted and *sort-decreasing* rewrite theories. If for all specializations ν $LS(\nu(\ell)) \geq LS(\nu(r))$ then we say that the OS-rule $\ell \to r$ if c is *sort-decreasing*. We call an OSRT $\mathcal{R} = (\Sigma, B, R)$ *sort-decreasing* if all rules in R are so. Due to our assumption that the equations B are sort-preserving and the B-coherence assumption, sort-decreasingness is stable under B-equivalence classes.

Remark 3. Our definition of sort-decreasing conditional rule does *not* impose anything to the conditional part of the rules. In this paper, we need sort-decreasingness to ensure *monotonicity* of conditional rewriting (see Proposition 1). This holds without any further restriction on the conditions of the rules.

Thanks to the stability of sort-decreasing rules under B-equality ensured by the assumptions on B we then have:

Proposition 1. [8] *Let $\mathcal{R}$ be a sort-decreasing OSRT, $t, u, v \in \mathcal{T}(\Sigma, \mathcal{X})$ and $p \in \mathcal{P}os(t)$. If $t = t[u]_p$ and $u \to v$, then $t[u]_p \to t[v]_p$.*

Without sort-decreasingness, this important result does not hold (see [8]). This assumption is essential in our proof of the following result.

Theorem 2. *Let $\mathcal{R}$ be a sort-decreasing deterministic 3-OSRT. If $\mathcal{R}$ is operationally terminating, then it is quasi-decreasing.*

Thus, quasi-decreasingness characterize operational termination of order-sorted, sort-decreasing rewrite theories.

Corollary 1. *A sort-decreasing deterministic 3-OSRT $\mathcal{R}$ is operationally terminating if and only if it is quasi-decreasing.*

4 Computing with Normal Rewrite Theories

Definition 2 (Irreducible forms and weak termination). *Let $\mathcal{R}$ be an OSRT and s, t be terms. We say that t is irreducible if, for any term u, $t \not\to_{\mathcal{R}} u$. $\mathsf{Irr}(\mathcal{R})$ (resp. $\mathsf{GIrr}(\mathcal{R})$) is the set of irreducible terms (resp. ground terms) of $\mathcal{R}$.*

If s rewrites to an irreducible term t, we say that s has a (not necessarily unique) irreducible form t, denoted $s \to\!\!\!\!\to t$. If every term s has an irreducible form, i.e., $s \to\!\!\!\!\to t$ for some irreducible term t, then $\mathcal{R}$ is called weakly terminating.

Terminating OSRTs are weakly terminating (in general, the opposite is not true).

Definition 3 (Normal form, weak normalization). *A term t is called a* normal form *if it is irreducible and operationally 1-terminating. Let $\mathsf{NF}(\mathcal{R})$ (resp. $\mathsf{GNF}(\mathcal{R})$) be the set of normal forms (resp. ground normal forms) of $\mathcal{R}$.*

If $s \to\!\!\!\!\to t$ and t is a normal form, we then write $s \to^! t$ and call t a normal form *of s. If every term s has a normal form, i.e., $s \to^! t$ for some normal form t, then $\mathcal{R}$ is called* weakly operationally terminating *(or* weakly normalizing*).*

Remark 4 (Notation). If $\mathcal{R}$ is confluent and weakly operationally terminating, then we write $t \to^!_{\mathcal{R}} u$ for $t \to\!\!\!\!\to_{\mathcal{R}} u$, denote such a u by $u = t!_{\mathcal{R}}$ or $u = can_{\mathcal{R}}(t)$, and call it *the $\mathcal{R}$-canonical form* of t which is unique up to B-equality.

Note that $\to\!\!\!\!\to_{\mathcal{R}/B} \supseteq \to^!_{\mathcal{R}/B}$ and $\mathsf{NF}(\mathcal{R}) \subseteq \mathsf{Irr}(\mathcal{R})$ (this inclusion can be strict!).

Example 1. The one-step rewrite relation for $a \to b$ if $a \to c$ (a single rule OSRT) is empty. Hence, a is irreducible. However, a is *not* a normal form: every attempt to prove a reduction step on a starts an infinite proof tree.

There can also be *reducible* terms that are *not* operationally 1-terminating.

Example 2. Term $f(a)$ is not operationally 1-terminating in the 2-CTRS $\mathcal{R}$:

$$g(a) \to c(b) \tag{1}$$
$$b \to f(a) \tag{2}$$
$$f(x) \to x \text{ if } g(x) \to c(y) \tag{3}$$

Since $g(a) \to c(b)$, we have $f(a) \to a$ by means of a finite proof tree. However, since the evaluation of the condition could *continue* beyond $c(b)$

$$g(a) \to c(\underline{b}) \to c(f(a))$$

and the term $f(a)$ can start a new (deep) proof tree, we also have an infinite (well-formed) proof tree for the goal $f(a) \to u$ with u arbitrary.

Remark 5. Note that $\mathcal{R}$ in Example 2 is *terminating*, i.e., there is no infinite rewrite sequence $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots$. This is easy to see, because the *underlying TRS* $\mathcal{R}_u = \{\ell \to r \mid \ell \to r \text{ if } c \in \mathcal{R}\}$ is clearly terminating.

Definition 4 (Normal and strongly deterministic rewrite theory). *A deterministic OSRT $\mathcal{R}$ is called* normal *(resp.* ground normal*) if the set* $\mathsf{Irr}(\mathcal{R})$ *(resp. the set* $\mathsf{GIrr}(\mathcal{R})$*) is operationally terminating, i.e., every irreducible (ground) term is a (ground) normal form:* $\mathsf{Irr}(\mathcal{R}) = \mathsf{NF}(\mathcal{R})$ *(resp.* $\mathsf{GIrr}(\mathcal{R}) = \mathsf{GNF}(\mathcal{R})$*).*

A normal OSRT $\mathcal{R} = (\Sigma, B, R)$ is called strongly deterministic *if for each $\ell \to r$ if $s_1 \to t_1, \ldots, s_n \to t_n$ in R, and each substitution θ such that $\theta(x) \in \mathsf{NF}(\mathcal{R})$ for each $x \in \mathcal{X}$, we have: $\theta(t_1), \ldots, \theta(t_n) \in \mathsf{NF}(\mathcal{R})$.*

The B-coherence assumption then gives us:

Proposition 2. *If a strongly deterministic 3-OSRT $\mathcal{R}$ is (ground) confluent and weakly normalizing, then $\mathcal{R}$ is (ground) normal.*

Remark 6. Ground normality is the minimum prerequisite for executability. For ground normal and ground confluent deterministic 3-OSRT $\mathcal{R}$, each ground term t has *at most* one normal form up to B-equality and the process $t \mapsto [t!_{\mathcal{R}}]_B$ defines a recursive partial function, since $\mathcal{R}$ need not even be weakly terminating.

In order to prove that a strongly deterministic OSRT $\mathcal{R} = (\Sigma, B, R)$ is ground normal, we can proceed as follows:

1. Identify a subsignature of constructors Ω with nonempty sorts such that the rules in R decompose as a disjoint union $R_{(\Sigma-\Omega)} \cup R_\Omega$, where the R_Ω have only Ω terms in their rules and conditions, and each $\ell \to r$ if $s_1 \to t_1, \ldots, s_n \to t_n$ in $R_{(\Sigma-\Omega)}$ has $l = f(t_1, \ldots, t_n)$ for some $f \in \Sigma - \Omega$. We also assume that the axioms B decompose as a disjoint union $B_{(\Sigma-\Omega)} \cup B_\Omega$ with the B_Ω involving only Ω terms, and the $B_{(\Sigma-\Omega)}$ *not* Ω-equations. This yields an ORST inclusion $\mathcal{R}_\Omega \subseteq \mathcal{R}$, with $\mathcal{R}_\Omega = (\Omega, B_\Omega, R_\Omega)$.

2. Prove (by inductive theorem proving) that for all defined symbols $f \in \Sigma - \Omega$, say with rank $f : s_1 \cdots s_n \longrightarrow s$, the following inductive property holds:

$$\forall x_1 \in \mathcal{T}_{\Omega_{s_1}}, \ldots, x_n \in \mathcal{T}_{\Omega_{s_n}}, \exists y f(x_1, \ldots, x_n) \rightarrow^1_{\mathcal{R}} y$$

Then if $\mathcal{R}_\Omega$ is *operationally terminating*, $\mathcal{R}$ is *ground normal* and, furthermore, $\mathsf{GNF}(\mathcal{R}) \subseteq \mathcal{T}_\Omega$. That is, an inductive proof of ground reducibility w.r.t. the constructors shows that $t \in \mathcal{T}(\Sigma)$ is a ground normal form iff:

1. $t \in \mathcal{T}_\Omega$; and
2. $t \in \mathsf{GNF}(\mathcal{R}_\Omega)$.

The assumptions on B give us:

Proposition 3. *Let $\mathcal{R} = (\Sigma, B, R)$ be a normal, sort-decreasing, confluent, strongly deterministic 3-OSRT such that R is finite. If $\mathcal{R}$ is weakly operationally terminating, then the function $t \mapsto [t!_{\mathcal{R}}]_B$ is* total recursive *and preserves sorts.*

Note that, otherwise, if $\mathcal{R}$ is confluent but *not* weakly operationally terminating, then the function $t \mapsto [u]_B$ with $t \twoheadrightarrow u$ *may not be recursive*, even if each t has an *irreducible* form. Implicit in Proposition 3 is the fact that, under such conditions plus the assumptions on B, when we intepret each $\ell \rightarrow r$ if $s_1 \rightarrow t_1, \ldots, s_n \rightarrow t_n$ in R as a *conditional equation* $\ell = r$ if $s_1 = t_1, \ldots, s_n = t_n$, normal forms define an *algebra* $\mathcal{C}_{\Sigma/R,B}$, called the *canonical term algebra* of $\mathcal{R}$. Specifically, for each sort s we define $\mathcal{C}_{\Sigma/R,B,s} = \mathsf{GNF}(\mathcal{R})/{=_B} \cap \mathcal{T}_{\Sigma/B}$, that is, the set of B-equivalence classes of ground normal forms of sort s, and, for each $f : s_1 \cdots s_n \longrightarrow s$ in Σ its intepretation in $\mathcal{C}_{\Sigma/R,B}$ maps each tuple $([t_1]_B, \ldots, [t_n]_B)$ with $[t_i]_B \in \mathcal{C}_{\Sigma/R,B,s_i}$ to the B-equivalence class $[f(t_1, \ldots, t_n)!_{\mathcal{R}}]_B$, which is well-defined and unique because of confluence, sort-decreasingness and B-coherence. The agreement between the operational semantics of $\mathcal{R}$ when terms are normalized by rewriting, and the mathematical semantics of $\mathcal{R}$ when its rules are interpreted as conditional equations can then be expressed for such general OSRTs as follows:

Corollary 2. *For $\mathcal{R} = (\Sigma, B, R)$ a sort-decreasing, confluent and weakly operationally terminating strongly deterministic 3-OSRT, the canonical term algebra $\mathcal{C}_{\Sigma/\mathcal{R}}$ is a* computable algebra. *Furthermore, $\mathcal{T}_{\Sigma/R \cup B} \simeq \mathcal{C}_{\Sigma/R,B}$.*

The general method to compute the normal form $t!_{\mathcal{R}}$ of a term t described in the proof of Proposition 3 is somewhat complex, and can be computationally expensive. It is therefore useful to seek somewhat less general conditions under which we can compute normal forms. We consider two such conditions, which can be executed in Maude in a straightforward way.

The first case is that of a strongly deterministic 3-OSRT that is sort-decreasing, ground confluent, 1-terminating, and ground weakly terminating and has a finite number of rewrite rules. Under such conditions, the `search` command in Maude asking for the fully-reduced first result for the given input ground term will compute such a normal form. This assumes that the rules in the theory are expressed as *rules* in a Maude system module and *not as equations* in a functional module, even though the module does indeed have a *functional semantics*.

A simple theory transformation, easily definable in Maude's `META-LEVEL` module, can transform the given functional module into its associated system module. Let us illustrate this general method with an example. Note that in this example the set B of axioms is empty. The functional module `fmod WEAK-NORM endfm` expresses the rewrite rules R as conditional equations, whereas the system module `mod WEAK-NORM endm` expresses them explicitly as rewrite rules.

```
fmod WEAK-NORM is
  protecting BOOL .
  sorts Nat Nat? .
  subsort Nat < Nat? .
  op 0 : -> Nat .              op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .    op even : Nat -> Bool .
  ops f g : Nat? -> Nat? .
  vars N M : Nat .
  eq N + 0 = N .                eq N + s(M) = s(N + M) .
  eq even(0) = true .           eq even(s(0)) = false .
  eq even(s(s(N))) = even(N) .  eq g(N) = N .
  eq f(N) = N + N .
  ceq f(N) = g(f(N)) if true := even(N) .
endfm
```

This module is sort-decreasing, weakly terminating and ground confluent. By the technique presented in Section **??**, we can prove it normal. Giving to Maude the term `f(0)` for evaluation leads to non-terminating behavior. That is, the usual operational semantics for evaluating operationally terminating confluent theories cannot be relied upon to compute normal forms. This problem can be solved by transforming the above functional module into a system module, that is, by transforming equations into rules, and using Maude's `search` command:

```
mod WEAK-NORM is
  protecting BOOL .
  sorts Nat Nat? .
  subsort Nat < Nat? .
  op 0 : -> Nat .                op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .      op even : Nat -> Bool .
  ops f g : Nat? -> Nat? .
  vars N M : Nat .
  rl N + 0 => N .                rl N + s(M) => s(N + M) .
  rl even(0) => true .           rl even(s(0)) => false .
  rl even(s(s(N))) => even(N) .  rl g(N) => N .
  rl f(N) => N + N .
  crl f(N) => g(f(N)) if even(N) => true .
endm
```

The normal form of a term can then be obtained by searching for the *first* result of a *terminating* computation from the given term. By confluence such a result is unique up to B-equality, exists by weak operational termination, and can be found by search without risk of looping thanks to 1-termination:

```
Maude> search [1] f(0) =>! N:Nat .
search in WEAK-NORM : f(0) =>! N .
Solution 1 (state 5)
states: 9  rewrites: 12 in 0ms cpu (0ms real) (44943 rewrites/second)
N --> 0
Maude> search [1] f(s(s(0))) =>! N:Nat .
search in WEAK-NORM : f(s(s(0))) =>! N .
Solution 1 (state 14)
states: 20  rewrites: 35 in 0ms cpu (0ms real) (55118 rewrites/second)
N --> s(s(s(s(0))))
Maude> search [1] f(s(s(s(s(0))))) =>! N:Nat .
search in WEAK-NORM : f(s(s(s(s(0))))) =>! N .
Solution 1 (state 27)
states: 35  rewrites: 70 in 1ms cpu (1ms real) (57189 rewrites/second)
N --> s(s(s(s(s(s(s(0)))))))
```

The second case where execution of a weakly operationally terminating deterministic OSRT can be achieved using execution mechanisms already available in Maude and yields again a full agreement between operational and mathematical semantics is the one of context-sensitive OSRTs under some reasonable assumptions. A *context-sensitive* [6] OSRT is a four-tuple $\mathcal{R} = (\Sigma, B, R, \mu)$, where (Σ, B, R) is an OSRT, and μ maps each $f : s_1 \cdots s_n \longrightarrow s$ in Σ to a subset $\mu(f) \subseteq \{1, \ldots, n\}$ of the argument positions of f under which rewriting is allowed. The operational semantics of context-sensitive OSRTs is defined by restricting the inference system of Figure 1 with the single restriction that, in the (Cong) Rule, i with $1 \leq i \leq k$ must furthermore satisfy $i \in \mu(f)$.

The Lemma below states the required conditions on $\mathcal{R} = (\Sigma, B, R, \mu)$ yielding the desired agreement between operational and mathematical semantics. This result relies on the notion of μ-sufficient completeness and of the algebra $\mathcal{C}^{\mu}_{\mathcal{R}}$ of term in μ-normal form (see [5]).

Lemma 1. *If $\mathcal{R}$ is a confluent, sort decreasing and strongly deterministic context-sensitive 3-OSRT $\mathcal{R} = (\Sigma, B, R, \mu)$, which is μ-operationally terminating and μ-sufficiently complete for $\Omega \subseteq \Sigma$ a subsignature of free constructors modulo B, then:*

1. *$\mathcal{R}$ is ground weakly operationally terminating.*
2. *$\mathcal{C}^{\mu}_{\mathcal{R}} \restriction_{\Omega} = \mathcal{T}_{\Omega/B}$.*
3. *For each $t \in \mathcal{T}_{\Sigma}$, $t!_{\mathcal{R},B} = t!^{\mu}_{\mathcal{R},B}$, that is, the normal form and the μ-normal form of t (which can be computed by Maude's* reduce *command)* coincide.
4. *$\mathcal{T}_{\Sigma/E \cup B} \simeq \mathcal{C}^{\mu}_{E/B}$ (agreement between operational and denotational semantics).*

Under the assumptions of Lemma 1, we compute normal forms as follows: since Maude supports the execution of confluent context-sensitive 3-OSRTs $\mathcal{R} = (\Sigma, B, R, \mu)$ specified as functional modules, we just use `reduce` to compute normal μ-forms, which under the assumptions in Lemma 1 are also ordinary normal forms in the underlying OSRT (Σ, B, R). We can illustrate these ideas with the following example of a context-sensitive 3-OSRT in Maude:

```
fmod FACTORIAL is
  protecting NAT .
  op monus : Nat Nat -> Nat .
  op _~_ : Nat Nat -> Bool [comm] .
  op [_,_,_] : Bool Nat Nat -> Nat [strat (1 0)] .
  op fact : Nat -> Nat .
  vars N M : Nat .
  eq monus(s(N),s(M)) = monus(N,M) .
  ceq monus(N,M) = N if M := 0 .
  ceq monus(N,M) = 0 if N := 0 .
  eq N ~ N = true .
  eq s(N) ~ s(M) = N ~ M .
  eq 0 ~ s(N) = false .
  eq [true,N,M] = N .
  eq [false,N,M] = M .
  eq fact(N) = [(N ~ 0),s(0),N * fact(monus(N,s(0)))] .
endfm
```

This theory, though ground confluent, is cleary non-terminating because of the last equation. Here, μ does not restrict any argument positions, except for the if-then-else operator $[\_,\_,\_]$, where $\mu([\_,\_,\_]) = \{1\}$, as specified by the `strat` attribute. It is, however, operationally μ-terminating and has `0` and `s`, and `true`, `false` as free constructors. Here are some evaluations:

```
Maude> red fact(2) .
reduce in FACTORIAL : fact(2) .
rewrites: 15 in 0ms cpu (0ms real) (192307 rewrites/second)
result NzNat: 2
Maude> red fact(3) .
reduce in FACTORIAL : fact(3) .
rewrites: 21 in 0ms cpu (0ms real) (10500000 rewrites/second)
result NzNat: 6
Maude> red fact(4) .
reduce in FACTORIAL : fact(4) .
rewrites: 27 in 0ms cpu (0ms real) (692307 rewrites/second)
result NzNat: 24
Maude> red fact(5) .
reduce in FACTORIAL : fact(5) .
rewrites: 33 in 0ms cpu (0ms real) (358695 rewrites/second)
result NzNat: 120
```

We end this section with the following result that, though well-known (see, e.g., [12]), has an easier proof with a rewrite theory with axioms B of associtivity and identity for strings. In some sense this result shows how wild the beasts in the general menagerie of OSRTs can be, and illustrates the need for notions such as that of normal theory to obtain reasonable computational behaviors.

Theorem 3. *There is a 2-OSRTs $\mathcal{R}$ and a sort s such that the set $\mathsf{Irr}(\mathcal{R})_s \subseteq \mathcal{T}(\mathcal{F},\mathcal{X})_s$ of $\mathcal{R}$-irreducible terms is* not recursively enumerable*, so it is not even* semi-decidable *if a term is $\mathcal{R}$-irreducible.*

5 Proving Order-Sorted Rewrite Theories Normal

1-operationally terminating rewrite theories are *normal*. The opposite is not true.

Example 3. The CTRS $\mathcal{R}$ in Example 2 is *not* 1-operationally terminating. However, $\mathcal{R}$ is normal: assume that there is a minimal irreducible term s having an infinite well-formed proof tree whose strict subterms are normal forms. Since f is the only symbol defined by a conditional rule, $s = f(t)$ for some normal form t. Since $f(t)$ is irreducible, the evaluation of the condition in the rule cannot succeed, i.e., $g(t)$ must be irreducible. Since t is a normal form, $g(t)$ cannot start any infinite well-formed tree. Contradiction.

Remark 7. As noticed in Remark 5, $\mathcal{R}$ in Example 2 is terminating. Since $\mathcal{R}$ is *not* 1-operationally terminating and a fortiori not operationally terminating, it follows from Example 3 that neither 1-operational termination nor operational termination of $\mathcal{R}$ follow from the termination and normality of $\mathcal{R}$.

An interesting feature in the treatment of innermost termination problems using the dependency pair approach [1] is that, since the variables in the right-hand side of the dependency pairs are in normal form, the rules which can be used to connect contiguous dependency pairs are usually a proper subset of the rules in the TRS. This leads to the notion of *usable rules* [1, Definition 32] which simplifies the proofs of innermost termination of rewriting.

In our analysis of normal rewrite theories we have a similar situation: when an irreducible term $t = f(t_1, \ldots, t_k)$ is tested to see whether it is a normal form, we know that all possible reductions derived from a proof $t \to x$ (for a fresh variable x) cause the evaluation of the conditional part c of some conditional rule $f(\ell_1, \ldots, \ell_k) \to r$ if c. Therefore, if we single out those rules that *can be* involved in any attempt to evaluate $\sigma(c)$ for some σ such that $t = \sigma(f(\ell_1, \ldots, \ell_k))$, we can obtain a more precise analysis. The notion of usable rule provides an upper, purely syntactic, approximation to the set of rules that eventually apply to a term t during any possible rewriting on t. We keep the original flavor of the original, unsorted notion in the following definition.

Definition 5 (Usable rules for a rewrite theory). *Let $\mathcal{R} = (\Sigma, B, R)$ be an OSRT. Let $RULES(\mathcal{R}, t)$ be the set of rules defining symbols occurring in t:*

$$RULES(\mathcal{R}, t) = \{\ell \to r \text{ if } c \in R \mid \exists p \in \mathcal{P}os(t), root(\ell) = root(t|_p)\}$$

Then, the set of usable rules *of $\mathcal{R}$ for t is:*

$$\mathcal{U}(\mathcal{R}, t) = RULES(\mathcal{R}, t) \cup \bigcup_{l \to r \text{ if } c \in RULES(\mathcal{R}, t)} \mathcal{U}(\mathcal{R}', r) \cup \bigcup_{s_i \to t_i \in c} \mathcal{U}(\mathcal{R}', s_i)$$

where $\mathcal{R}' = \mathcal{R} - RULES(\mathcal{R}, t)$.

That is: we consider both unconditional and conditional rules and add the rules that could be *used* to evaluate the conditions in the rules. Since we are dealing

with OSRTs $\mathcal{R} = (\Sigma, B, R)$, rewriting happens *modulo* B. This raises the issue of whether the above definition of usable rules is overly syntactic, that is, not stable under B-equality. The key issue is whether in the (Repl) rule in the inference system of Figure 1 the top symbol of the redex u coincides with that of the lefthand side l. This is the case by requiring the axioms B to be as follows:

$$B = \bigcup_{f:[s_1]\cdots[s_n]\to[s]\in\Sigma} B_f$$

where B_f is a set of equations $u = v$ with $u, v \in \mathcal{T}(\{f\}, \mathcal{X}) - \mathcal{X}$, i.e., only symbol f is allowed (and must) to occur in the equations belonging to B_f. Associativity and commutativity axioms satisfy this requirement which can even be made to work for identity axioms by perfoming the semantics-preserving transformation described in [3]. Now we can give the main result of this section. For an OSRT $\mathcal{R} = (\Sigma, B, R)$, we say that B preserves the $\mathcal{R}$-normal forms if for all $\mathcal{R}$-normal forms t, if $t =_B u$, then u is an $\mathcal{R}$-normal form. *B-coherence*, which is a usual requirement for working OSRTs, implies this property. By $\mathcal{R}_C$ we denote the OSRT obtained as the union of $\mathcal{U}(\mathcal{R}, s)$ for all *lhs*'s conditions in the rules of $\mathcal{R}$:

$$\mathcal{R}_C = \bigcup_{\ell\to r \text{ if } c} \bigcup_{s\to t\in c} \mathcal{U}(\mathcal{R}, s)$$

Theorem 4. *A deterministic 3-OSRT $\mathcal{R} = (\Sigma, B, R)$ is normal if B preserves the $\mathcal{R}$-normal forms and $\mathcal{R}_C$ is operationally terminating.*

Example 4. Consider the functional module `WEAK-NORM` in Section 4. Here, $\mathcal{R}_C$ is the unconditional subOSRT consisting of the rules defining `even`. Note that $\mathcal{R}_C$ has no conditional rule and is clearly terminating, hence operationally terminating. We conclude that, as claimed, `WEAK-NORM` is a normal OSRT.

Now we show that Theorem 4 does *not* characterize normality of OSRTs:

Example 5. Consider the following deterministic 1-CTRS:

$$\begin{array}{ll} a \to b & f(x) \to x \text{ if } c \to d, a \to c \\ b \to a & \end{array}$$

Every term $f(t)$ is *irreducible* and also a normal form because the unsatisfiable condition $c \to d$ prevents the looping evaluation of the condition $a \to c$. However, $\mathcal{R}_C = \{a \to b, b \to a\}$ is not (operationally) terminating.

6 Conclusions and Future Work

The results presented in this paper can be viewed from two complementary perspectives: one more theoretical, and another more practical. At the theoretical level, we have investigated parts of the *terra incognita* of conditional term rewriting by asking and providing precise answers to innocent-sounding questions such

as: what is a normal form? How can normal forms be effectively computed? How should the notion of weakly normalizing system be understood in the conditional case? How can good executability properties be ensured for a theory? There is, however, a more practical aspect to all these results. It consists in taking to heart the idea that rewrite theories are an excellent framework for declarative programming and formal specification and verification. From this second perspective, the questions asked and answered include: what is the most general notion possible of a conditional rule-based program for which normal forms can be computed? What is the appropriate term normalization operational semantics? How can it be made more efficient? What are the most general possible requirements under which conditional functional programs can be given an initial algebra semantics which *fully agrees* with their operational semantics?

Future work should further investigate proof methods and supporting tools for all the properties discussed here. For example, although the characterization of the operation termination of an OSRT in terms of quasi-decreasingness offers in principle a complete proof method, we are currently investigating a far-reaching generalization to the conditional case of the dependency pair method that seems considerably more effective for mechanizing actual proofs. In general, the development of *intrinsic methods* for proving both strong and weak operational termination of OSRTs seems both quite attractive and sorely needed.

References

1. T. Arts and J. Giesl. Termination of Term Rewriting Using Dependency Pairs. *Theoretical Computer Science*, 236(1–2):133–178, 2000.
2. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. All About Maude – A High-Performance Logical Framework. LNCS 4350, 2007.
3. F. Durán, S. Lucas, J. Meseguer, Termination Modulo Combinations of Equational Theories, In *Proc of FroCoS'09*, LNAI 5749:246–262, Springer-Verlag, 2009.
4. J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
5. J. Hendrix and J. Meseguer On the Completeness of Context-Sensitive Order-Sorted Specifications In *Proc. of RTA'07*, LNCS 4533:229-245, Springer-Verlag, 2007.
6. S. Lucas. Context-Sensitive Computations in Functional and Functional Logic Programs. *Journal of Functional and Logic Programming*, 1998(1):1–61, 1998.
7. S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95:446–453, 2005.
8. S. Lucas and J. Meseguer. Order-Sorted Dependency Pairs. In *Proc. of PPDP'08*, pp. 108-119, ACM Press, 2008.
9. J. Meseguer. Membership algebra as a logical framework for equational specification. In *Proc. of WADT'97*, LNCS 1376:18–61. Springer-Verlag, 1998.
10. J. Meseguer. On Coherence. *To appear* 2014.
11. E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, Apr. 2002.
12. TeReSe, Term Rewriting Systems, Cambridge University Press, 2003.

2D Dependency Pairs for Proving Operational Termination of CTRSs*

Salvador Lucas[1,2] and José Meseguer[2]

[1] DSIC, Universitat Politècnica de València, Spain
[2] CS Dept. University of Illinois at Urbana-Champaign, IL, USA

Abstract. The notion of *operational termination* captures nonterminating computations due to subsidiary processes that are necessary to issue a *single* 'main' step but which often remain 'hidden' when the main computation sequence is observed. This highlights *two dimensions* of nontermination: one for the infinite sequencing of computation steps, and the other that concerns the proof of some single steps. For conditional term rewriting systems (CTRSs), we introduce a new *dependency pair framework* which exploits such *bidimensional* nature of conditional rewriting (rewriting steps + satisfaction of the conditions as reachability problems) to obtain a powerful framework for proving operational termination of CTRSs.

Keywords: Conditional term rewriting, dependency pairs, program analysis, operational termination.

1 Introduction

Assume that we have an *interpreter* for a logic $\mathcal{L}$, i.e., an *inference machine* that, given a theory $\mathcal{S}$ and a goal formula φ, will try to incrementally build a proof tree for φ. Intuitively, we call $\mathcal{S}$ *terminating* if for any φ the interpreter either finds a proof in finite time, or fails in all possible attempts also in finite time. The notion of *operational termination* captures this fact, meaning that, given an initial goal, an interpreter will either succeed in finite time in producing a closed proof tree, or will fail in finite time, not being able to close or extend further any of the possible proof trees, after exhaustively searching all such proof trees [11]. Operational termination captures a *'vertical' dimension* of the termination behavior which is missing in the usual definition of termination of relations as *well-founded*, i.e., "without infinite reduction sequences" (the 'horizontal' dimension).

Available tools for proving operational termination of conditional rewriting (e.g., AProVE [9] or VMTL [15]) rely on *transformations* $\mathcal{U}$ that map each operational termination problem for the CTRS $\mathcal{R}$ into a termination problem for a

* Research partially supported by NSF grant CNS 13-19109. Salvador Lucas' research was developed during a sabbatical year at the CS Department of the UIUC and was also partially supported by Spanish MECD grant PRX12/00214, MINECO project TIN2010-21062-C02-02, and GV project PROMETEO/2011/052.

TRS $\mathcal{U}(\mathcal{R})$. Then, available methods for proving termination of $\mathcal{U}(\mathcal{R})$ are used. However, this transformational approach has strong limitations.

Example 1. Consider the following CTRS $\mathcal{R}$ [14, Example 8]

$$h(d) \to c(a) \tag{1}$$
$$h(d) \to c(b) \tag{2}$$
$$f(k(a), k(b), x) \to f(x, x, x) \tag{3}$$
$$g(x) \to k(y) \text{ if } h(x) \to d, h(x) \to c(y) \tag{4}$$

As reported in [14, Example 8], $\mathcal{U}(\mathcal{R})$ is *not* terminating. However, our methods will show that $\mathcal{R}$ is operationally terminating.

Most termination tools for proving termination of (variants of) rewriting with TRSs implement extensions or generalizations of the dependency pair framework [6, 7]. The main idea is the following: the rules $\ell \to r$ that are able to produce infinite sequences are those whose right-hand side r contains (possibly recursive) *function calls*. The calls associated to $\ell \to r$ are represented as new rules $u \to v$, that are collected in a new TRS $\mathsf{DP}(\mathcal{R})$ of *dependency pairs* (DPs); $\mathcal{R}$ and $\mathsf{DP}(\mathcal{R})$ determine *dependency chains* whose finiteness characterize termination of $\mathcal{R}$ [1].

In this paper, we generalize this approach to *deterministic* 3-*CTRSs*, which are the basis of rewriting-based languages like CafeOBJ [5] or Maude [3]. In Section 3, we show that computations starting from *minimal* operationally nonterminating terms can always follow a precise path where two sources of nontermination can be identified: infinite sequences of rewriting steps (an *horizontal* dimension), and infinitely many attempts to check the satisfaction of the conditions in the rules (a *vertical* dimension). Section 4 introduces a definition of dependency pairs that makes such a bidimensional nature of infinite computations explicit (we call them *2D-DPs*). The corresponding notion of chain of dependency pairs permits a completely *independent* treatment of both dimensions of the termination problems. In Section 5, we adapt the *Dependency Pair Framework* [6, 7] to mechanize proofs of operational termination of deterministic 3-CTRSs using 2D-DPs. In Example 8, we prove the operational termination of $\mathcal{R}$ in Example 1. Section 6 discusses related work and concludes.

2 Preliminaries

Recall from [13] the usual notions and notations regarding term rewriting and CTRSs. An (*oriented*) CTRS $\mathcal{R}$ is a pair $\mathcal{R} = (\mathcal{F}, R)$ where $\mathcal{F}$ is a signature and R a set of rules $\ell \to r$ if $s_1 \to t_1, \cdots, s_n \to t_n$, where the conditions $s_i \to t_i$ for $1 \leq i \leq n$ are intended to express the reachability of (instances of) t_i from (instances of) s_i. As usual, ℓ and r are called the left- and right-hand sides of the rule, and the sequence $s_1 \to t_1, \cdots, s_n \to t_n$ (often abreviated to c) is the *conditional part* of the rule. Rewrite rules $\ell \to r$ if c are classified according to the distribution of variables among l, r, and c, as follows: type 1,

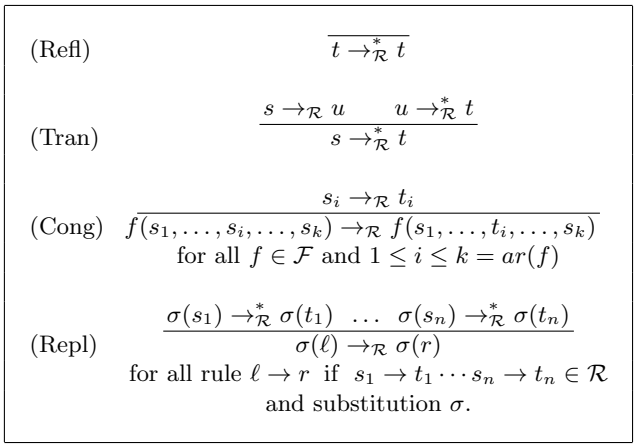


Fig. 1. Inference rules for conditional rewriting

if $\mathcal{V}ar(r) \cup \mathcal{V}ar(c) \subseteq \mathcal{V}ar(\ell)$; type 2, if $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(\ell)$; type 3, if $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(\ell) \cup \mathcal{V}ar(c)$; and type 4, if no restriction is given. An n-CTRS contains only rewrite rules of type $m \le n$. An oriented 3-CTRS $\mathcal{R}$ is called *deterministic* if for each rule $\ell \to r$ if $s_1 \to t_1, \ldots, s_n \to t_n$ in $\mathcal{R}$ and each $1 \le i \le n$, we have $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(l) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(t_j)$. Given $\mathcal{R} = (\mathcal{F}, R)$, we consider $\mathcal{F}$ as the disjoint union $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ of symbols $c \in \mathcal{C}$ (called *constructors*) and symbols $f \in \mathcal{D}$ (called *defined functions*), where $\mathcal{D} = \{root(l) \mid (l \to r \text{ if } c) \in R\}$ and $\mathcal{C} = \mathcal{F} - \mathcal{D}$. Terms $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $root(t) \in \mathcal{D}$ are called *defined* terms. $Pos_{\mathcal{D}}(t)$ is the set of positions p of subterms $t|_p$ such that $root(t|_p) \in \mathcal{D}$.

We say that a proof tree T is *closed* whenever it is finite and contains no open goals; it is *well-formed* if it is either an open goal, or a closed proof tree, or a derivation tree of the form $\frac{T_1 \quad \cdots \quad T_n}{G}$ where, for each j, T_j is itself well-formed, and there is $i \le n$ such that T_i is not closed, for any $j < i$ T_j is closed, and each of the T_{i+1},...,T_n is an open goal [11]. An infinite proof tree is *well-formed* if it is an ascending chain of well-formed finite proof trees. Intuitively, well-formed trees are the trees that an interpreter would incrementally build when trying to solve one condition at a time from left to right. We write $s \to_{\mathcal{R}} t$ (resp. $s \to^*_{\mathcal{R}} t$) iff there is a well-formed proof tree for $s \to_{\mathcal{R}} t$ (resp. $s \to^*_{\mathcal{R}} t$) using the inference system in Figure 1. The CTRS $\mathcal{R}$ is called *operationally terminating* if no infinite well-formed tree for a goal $s \to_{\mathcal{R}} t$ or $s \to^*_{\mathcal{R}} t$ exists.

3 Minimal operationally nonterminating terms in CTRSs

In the following, given a proof tree T, $root(T)$ is the formula (goal) at the root of the tree, and $left(G)$ is the left-hand side of goal G, where G is $s \to t$ or $s \to^* t$ for some terms s and t.

Definition 1 (Operationally nonterminating term). *Let $\mathcal{R}$ be a CTRS. A term t such that $left(root(T)) = t$ for an infinite well-formed proof tree T is called* operationally nonterminating. *If there is no infinite well-formed proof tree T such that $left(root(T)) = t$, then we call t* operationally terminating.

Definition 2 (Minimality). *Let $\mathcal{R}$ be a CTRS. An operationally nonterminating term t is called* minimal *if every strict subterm u of t (i.e., $t \rhd u$) is* operationally terminating. *Let $\mathcal{T}_{op\text{-}\infty}$ be the set of minimal operationally nonterminating terms associated to $\mathcal{R}$.*

The following lemma shows that operationally nonterminating terms always contain a *minimal* operationally nonterminatin term.

Lemma 1. *Let $\mathcal{R} = (\mathcal{F}, R)$ be a CTRS and $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. If s is operationally nonterminating, then there is a subterm t of s ($s \trianglerighteq t$) such that $t \in \mathcal{T}_{op\text{-}\infty}$.*

Proposition 1 below establishes that, for $t \in \mathcal{T}_{\text{op-}\infty}$, there is a precise way for an infinite computation to proceed. Roughly speaking, a rule $\ell \to r$ if $\bigwedge_{i=1}^{n} s_i \to t_i$ must be used to try a *root-step* on a reduct of t. Then, there is a minimal operationally nonterminating subterm which is either (1) an instance of a nonvariable subterm of the *right-hand side* r of the rule (the infinite computation continues through the *horizontal* dimension), or (2) an instance of a nonvariable subterm of one of the *left-hand sides* s_i of a condition $s_i \to t_i$ (the infinite computation continues through the *vertical* dimension). Given a term t, $\mathcal{D}Subterm(\mathcal{R}, t) = \{t|_p \mid p \in \mathcal{P}os_{\mathcal{D}}(t)\}$ is the set of defined subterms of t with respect to rules in $\mathcal{R}$. Let $DRules(\mathcal{R}, t)$ be the set of (possibly conditional) rules in $\mathcal{R}$ defining $root(t)$ which *depend* on other defined symbols in $\mathcal{R}$:

$$DRules(\mathcal{R}, t) = \{\ell \to r \text{ if } c \in \mathcal{R} \mid root(\ell) = root(t), r \notin \mathcal{T}(\mathcal{C}, \mathcal{X})\}$$

The *dependency* is captured as $r \notin \mathcal{T}(\mathcal{C}, \mathcal{X})$ in the definition. For each $v \in \mathcal{D}Subterm(\mathcal{R}, r)$, $DRules(\mathcal{R}, v)$ contains the rules that *will (eventually) be used* in root steps in the *immediate* continuation of the infinite computation in the *horizontal* dimension (starting from an instance of v). With regard to the *vertical* dimension, given a term t, the set of 'proper' conditional rule defining $root(t)$ is:

$$Rules_C(\mathcal{R}, t) = \{\ell \to r \text{ if } \bigwedge_{i=1}^{n} s_i \to t_i \in \mathcal{R} \mid root(\ell) = root(t), n > 0\}$$

These are the rules involved in transitions of computations to *upper* levels. We let $URules(\mathcal{R}, t) = DRules(\mathcal{R}, t) \cup Rules_C(\mathcal{R}, t)$ to be the set of used rules.

Proposition 1. *Let $\mathcal{R}$ be a deterministic 3-CTRS. Then, for all $t \in \mathcal{T}_{op\text{-}\infty}$, there exist $\alpha : \ell \to r$ if $\bigwedge_{i=1}^{n} s_i \to t_i$ and a substitution σ such that $t \xrightarrow{>\Lambda}{}^{*} \sigma(\ell)$, and there is a term v such that $\ell \not\trianglerighteq v$, $\sigma(v) \in \mathcal{T}_{op\text{-}\infty}$ and either*

1. *$\alpha \in DRules(\mathcal{R}, t)$, for all $1 \le i \le n$, $\sigma(s_i)$ is operationally terminating and $\sigma(s_i) \to^{*} \sigma(t_i)$, and $v \in \mathcal{D}Subterm(\mathcal{R}, r)$ is such that $URules(\mathcal{R}, v) \neq \emptyset$, or*

2. $\alpha \in \mathit{Rules}_C(\mathcal{R}, t)$, *there is* i, $1 \leq i \leq n$ *such that* $\sigma(s_j)$ *is operationally terminating and* $\sigma(s_j) \to^* \sigma(t_j)$ *for all* j, $1 \leq j < i$, *and* $v \in \mathcal{DS}\mathit{ubterm}(\mathcal{R}, s_i)$ *is such that* $\mathit{URules}(\mathcal{R}, v) \neq \emptyset$.

Remark 1. In the following we do *not* impose that the domain of the substitutions be finite. This is usual practice in the dependency pair approach, where a single substitution is used to instantiate an infinite number of variables coming from renamed versions of the dependency pairs (see below).

The next result formalizes a *bidimensional* view of infinite computations starting from minimal operational nonterminating terms: they can be viewed as a path over $\mathbb{N} \times \mathbb{N}$, where each bidimensional point (x_i, y_i) is labeled with a rule α_i.

Theorem 1. *Let* $\mathcal{R} = (\mathcal{F}, R)$ *be a deterministic* 3*-CTRS and* $t \in \mathcal{T}_{op\text{-}\infty}$. *There is a substitution* σ *and an infinite sequence* $\{(x_i, y_i, \alpha_i)\}_{i \in \mathbb{N}}$ *of triples* $(x_i, y_i, \alpha_i) \in \mathbb{N} \times \mathbb{N} \times R$ *such that, for all* $i \geq 0$, $x_{i+1} + y_{i+1} = x_i + y_i + 1$ *and*

1. $x_0 = y_0 = 0$, $\alpha_0 \in \mathit{URules}(\mathcal{R}, t)$ *and* $t \xrightarrow{>\Lambda}{}^* \sigma(\ell_0)$.
2. *For all* $i \geq 0$, *and* $\alpha_i : \ell_i \to r_i$ if $\bigwedge_{j=1}^{n_i} s_j^i \to t_j^i \in R$, *we have* $\sigma(\ell_i) \in \mathcal{T}_{op\text{-}\infty}$; *furthermore, there is a term* v_i *such that* $\ell_i \not\trianglerighteq v_i$, $\sigma(v_i) \in \mathcal{T}_{op\text{-}\infty}$, $\sigma(v_i) \xrightarrow{>\Lambda}{}^* \sigma(\ell_{i+1})$, $\alpha_{i+1} \in \mathit{URules}(v_i)$, *and*
   (a) *If* $x_{i+1} = x_i + 1$, *then* $v_i \in \mathcal{DS}\mathit{ubterm}(\mathcal{R}, r_i)$ *and* $\alpha_i \in \mathit{DRules}(\mathcal{R}, \ell_i)$.
   (b) *If* $y_{i+1} = y_i + 1$, *then there is* j, $1 \leq j \leq n_i$ *s.t.* $v_i \in \mathcal{DS}\mathit{ubterm}(\mathcal{R}, s_j^i)$ *and* $\alpha_i \in \mathit{Rules}_C(\mathcal{R}, \ell_i)$.

Example 2. Consider the following deterministic 3-CTRS $\mathcal{R}$:

$$g(a) \to c(b) \tag{5}$$
$$b \to f(a) \tag{6}$$
$$f(x) \to y \text{ if } g(x) \to c(y) \tag{7}$$

Figure 2 shows the representation of the computation starting from $f(a) \in \mathcal{T}_{\text{op-}\infty}$ according to Theorem 1, where the coordinates (x_i, y_i) have been left implicit.

Remark 2. The *minimal* sequence $f(a) \to_{(7)} b \to_{(6)} f(a) \to_{(7)} b \to \cdots$ is also possible for $\mathcal{R}$ in Example 2. This is because $\sigma(g(x)) \to^* (c(y))$ for rule (7) is satisfied *without any reduction on* b if $\sigma(x) = a$ and $\sigma(y) = b$. The implicit assumption in the computation model of Proposition 1 is that only reachability conditions $\sigma(s_i) \to^* \sigma(t_i)$ that are free of any infinite computation are important to decide the application of a rule. This makes real sense in practice. And, of course, it is harmless for the correctness or completeness of our approach.

According to our discussion, the following definition establishes the subsets of rules that play a special role in computations starting from minimal terms.

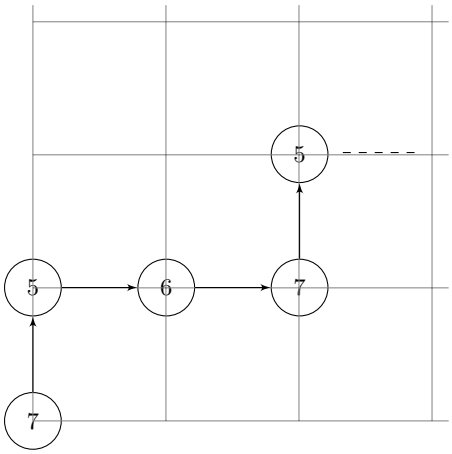


Fig. 2. Computations starting with $f(a)$ for $\mathcal{R}$ in Example 2

Definition 3. *The* dependent usable rules *for a CTRS $\mathcal{R}$ and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ are:*

$$\mathcal{DU}(\mathcal{R}, t) = DRules(\mathcal{R}, t) \cup \bigcup_{(l \to r \text{ if } c) \in DRules(\mathcal{R}, t)} \bigcup_{v \in \mathcal{DS}ubterm(\mathcal{R}, r)} \mathcal{DU}(\mathcal{R}^\bullet, v)$$

where $\mathcal{R}^\bullet = \mathcal{R} - DRules(\mathcal{R}, t)$. The set of minimal usable rules *of $\mathcal{R}$ for t is:*

$$\mathcal{MU}(\mathcal{R}, t) = URules(\mathcal{R}, t) \cup \bigcup_{(l \to r \text{ if } c) \in DRules(\mathcal{R}, t)} \bigcup_{v \in \mathcal{DS}ubterm(\mathcal{R}, r)} \mathcal{MU}(\mathcal{R}^\bullet, v)$$

Let $\overline{\mathcal{MU}}(\mathcal{R}, t) = \emptyset$ if $\mathcal{MU}(\mathcal{R}, t)$ is a TRS and $\overline{\mathcal{MU}}(\mathcal{R}, t) = \mathcal{MU}(\mathcal{R}, t)$ otherwise.

The following result shows that an infinite computation starting from a minimal operationally nonterminating term can either start an infinite (*horizontal*) *rewrite* sequence (possibly as part of the evaluation of one of the conditions of a rule) or else take infinitely many '*vertical*' shifts over the conditions in the rules.

Corollary 1. *Let $\mathcal{R}$ be a deterministic 3-CTRS and $t \in \mathcal{T}_{op\text{-}\infty}$. Then, the sequence $\{(x_i, y_i, \alpha_i)\}_{i \geq 0}$ associated to t according to Theorem 1 satisfies one of the following conditions. Either*

1. *There is $k \geq 0$, $\ell_k \to r_k$* if *$c_k \in \mathcal{R}$, and an infinite* 'horizontal' *sequence $\{(x_i, y_k, \alpha_i)\}_{i \geq k}$ such that for all $i \geq k$, $x_{i+1} = x_i + 1$ and $\alpha_i \in \bigcup_{v_k \in \mathcal{DS}ubterm(\mathcal{R}, r_k)} \mathcal{DU}(\mathcal{R}, v_k)$, or*
2. *For each $i \in \mathbb{N}$ such that $y_i > 0$ and $y_i = y_{i-1} + 1$, there is $k_i > i$ such that $y_{k_i} = y_i + 1$, and there is j_i, $1 \leq j_i \leq n_i$ such that $\alpha_{k_i - 1} \in \bigcup_{v_i \in \mathcal{DS}ubterm(\mathcal{R}, s_{ij_i})} \overline{\mathcal{MU}}(\mathcal{R}, v_i)$, with $n_{k_i - 1}$ conditions in the conditional part of the rule, satisfies $n_{k_i - 1} > 0$.*

In the following, we use Dependency Pairs to capture the nontermination behavior of computations with CTRSs.

4 2D Dependency Pairs for CTRSs

Given a signature $\mathcal{F}$ and $f \in \mathcal{F}$, we let $f^\sharp$ (often just capitalized, e.g., F) be a fresh symbol associated to f [1]. Let $\mathcal{F}^\sharp = \{f^\sharp \mid f \in \mathcal{F}\}$. For $t = f(t_1, \ldots, t_k) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, we write $t^\sharp$ to denote the *marked* term $f^\sharp(t_1, \ldots, t_k)$. Our Dependency Pairs for CTRSs are organized into two blocks. The *horizontal* block contains those pairs that correspond to rules issuing root steps in infinite rewrite sequences (Proposition 1, item 1):

$$\mathsf{DP}_H(\mathcal{R}) = \{\ell^\sharp \to v^\sharp \ \text{ if } \ c \mid \ell \to r \ \text{ if } \ c \in R, r \trianglerighteq v, \ell \not\trianglerighteq v, \mathit{DRules}(\mathcal{R}, v) \neq \emptyset\}$$

The *vertical* block contains pairs for shifting the infinite computation to the conditions of the rules (Proposition 1, item 2):

$$\begin{aligned}\mathsf{DP}_V(\mathcal{R}) = \{\ell^\sharp \to v^\sharp \ \text{ if } \ \textstyle\bigwedge_{j=1}^{k-1} s_j \to t_j \mid \ell \to r \ \text{ if } \ \bigwedge_{i=1}^{n} s_i \to t_i \in R,\\ \exists k, 1 \leq k \leq n, s_k \trianglerighteq v, \ell \not\trianglerighteq v, \mathit{URules}(\mathcal{R}, v) \neq \emptyset\}\end{aligned}$$

The subterms in the conditions of the rules that originate the pairs in $\mathsf{DP}_V(\mathcal{R})$ are collected in the following set, which we use below:

$$\mathsf{V}_C(\mathcal{R}) = \{v \mid \ell \to r \ \text{ if } \ \bigwedge_{i=1}^{n} s_i \to t_i \in R, \exists k, 1 \leq k \leq n, s_k \trianglerighteq v, \ell \not\trianglerighteq v, \mathit{URules}(\mathcal{R}, v) \neq \emptyset\}$$

We also have pairs to *connect* pairs in $\mathsf{DP}_V(\mathcal{R})$ (Corollary 1, item 2):

$$\mathsf{DP}_{VH}(\mathcal{R}) = \bigcup_{w \in \mathsf{V}_C(\mathcal{R})} \{\ell^\sharp \to v^\sharp \ \text{ if } \ c \mid \ell \to r \ \text{ if } \ c \in \overline{\mathcal{MU}}(\mathcal{R}, w), r \trianglerighteq v, \ell \not\trianglerighteq v, \mathit{URules}(\mathcal{R}, v) \neq \emptyset\}$$

Here is the definition of $2D$-Dependency Pairs for a CTRS.

Definition 4 ($2D$-Dependency Pairs). *The triple of* 2D-dependency pairs *(2D-DPs) for the CTRS $\mathcal{R}$ is* $\mathsf{DP}_{\mathsf{2D}}(\mathcal{R}) = (\mathsf{DP}_H(\mathcal{R}), \mathsf{DP}_V(\mathcal{R}), \mathsf{DP}_{VH}(\mathcal{R}))$.

Example 3. Consider the following 3-CTRS $\mathcal{R}$ in [13, Example 7.1.5]

$$\begin{aligned}
\mathsf{less}(x, \mathsf{0}) &\to \mathsf{false} && (8)\\
\mathsf{less}(\mathsf{0}, s(x)) &\to \mathsf{true} && (9)\\
\mathsf{less}(s(x), s(y)) &\to \mathsf{less}(x, y) && (10)\\
\mathsf{minus}(\mathsf{0}, s(y)) &\to \mathsf{0} && (11)\\
\mathsf{minus}(x, \mathsf{0}) &\to x && (12)\\
\mathsf{minus}(s(x), s(y)) &\to \mathsf{minus}(x, y) && (13)\\
\mathsf{quotrem}(\mathsf{0}, s(y)) &\to \mathsf{pair}(\mathsf{0}, \mathsf{0}) && (14)\\
\mathsf{quotrem}(s(x), s(y)) &\to \mathsf{pair}(\mathsf{0}, s(x)) && (15)\\
&\quad \text{if} \quad \mathsf{less}(x, y) \to \mathsf{true}\\
\mathsf{quotrem}(s(x), s(y)) &\to \mathsf{pair}(s(q), r) && (16)\\
&\quad \text{if} \quad \mathsf{less}(x, y) \to \mathsf{false}, \mathsf{quotrem}(\mathsf{minus}(x, y), s(y)) \to \mathsf{pair}(q, r)
\end{aligned}$$

we have:

$$
\begin{aligned}
\mathsf{DP}_H(\mathcal{R}):&\\
\mathsf{LESS}(s(x),s(y)) &\to \mathsf{LESS}(x,y) && (17)\\
\mathsf{MINUS}(s(x),s(y)) &\to \mathsf{MINUS}(x,y) && (18)\\
\mathsf{DP}_V(\mathcal{R}):&\\
\mathsf{QUOTREM}(s(x),s(y)) &\to \mathsf{LESS}(x,y) && (19)\\
\mathsf{QUOTREM}(s(x),s(y)) &\to \mathsf{QUOTREM}(\mathsf{minus}(x,y),s(y)) && (20)\\
&\quad \text{if} \quad \mathsf{less}(x,y)\to\mathsf{false}&&\\
\mathsf{QUOTREM}(s(x),s(y)) &\to \mathsf{MINUS}(x,y) && (21)\\
&\quad \text{if} \quad \mathsf{less}(x,y)\to\mathsf{false}&&\\
\mathsf{DP}_{VH}(\mathcal{R}):&\ \emptyset
\end{aligned}
$$

Example 4. Consider the 3-CTRS $\mathcal{R}$ in Example 2. We have:

$$
\begin{aligned}
\mathsf{DP}_H(\mathcal{R}) &= \{G(a)\to B\}\\
\mathsf{DP}_V(\mathcal{R}) &= \{F(x)\to G(x)\}\\
\mathsf{DP}_{VH}(\mathcal{R}) &= \{G(a)\to B, B\to F(a)\}
\end{aligned}
$$

4.1 Characterizing operational termination of CTRSs using 2D-DPs

An essential property of the dependency pair method is that it provides a *characterization* of termination of TRSs $\mathcal{R}$ as the absence of infinite (minimal) *chains of dependency pairs* [1, 7]. As we prove below, this is also true for deterministic 3-CTRSs when 2D-DPs are considered. First, we have to introduce a suitable notion of chain that can be used with 2D-DPs.

Definition 5 (Chain of pairs - Minimal chain). *Let $\mathcal{P},\mathcal{Q},\mathcal{R}$ be CTRSs. A $(\mathcal{P},\mathcal{Q},\mathcal{R})$-chain is a finite or infinite sequence of pairs $u_i\to v_i$ if $\bigwedge_{j=1}^{n_i} s_{ij}\to t_{ij}\in\mathcal{P}$, together with a substitution σ satisfying that, for all $i\geq 1$,*

1. *$\sigma(s_{ij})\to_{\mathcal{R}}^{*}\sigma(t_{ij})$ for all j, $1\leq j\leq n_i$ and*
2. *$\sigma(v_i)(\to_{\mathcal{R}}^{*}\circ\xrightarrow[\mathcal{Q}]{\Lambda}{}^{=}\,)^{*}\sigma(u_{i+1})$, where given a rule $\ell\to r$ if $\bigwedge_{j=1}^{n} s_j\to t_j\in\mathcal{Q}$, we write $s\xrightarrow[\mathcal{Q}]{\Lambda}{}^{=}t$ if either $s=t$ or there is a substitution θ such that $s=\theta(\ell)$, $t=\theta(r)$ and $\theta(s_i)\to_{\mathcal{R}}^{*}\theta(t_i)$ for all j, $1\leq j\leq n$ (note that the satisfaction of reachability constraints involves rewritings with $\mathcal{R}$).*

As usual, we assume that different occurrences of pairs do not share any variable (renaming substitutions are used if necessary). A $(\mathcal{P},\mathcal{Q},\mathcal{R})$-chain is called minimal *if for all $i\geq 1$, $\sigma(v_i)$ is $\mathcal{R}$-operationally terminating.*

Remark 3. Note that, if $\mathcal{P}$ and $\mathcal{R}$ are TRSs (without conditional rules) and $\mathcal{Q}=\emptyset$, Definition 5 specializes to the standard definition of chain of pairs in the Dependency Pair Framework for TRSs [7, Definition 3].

Now we provide a new characterization of operational termination of CTRSs.

Theorem 2 (Operational termination of CTRSs). *A deterministic 3-CTRS $\mathcal{R}$ is operationally terminating if and only if there is no infinite (minimal) $(\mathsf{DP}_H(\mathcal{R}), \emptyset, \mathcal{R})$-chain and there is no infinite (minimal) $(\mathsf{DP}_V(\mathcal{R}), \mathsf{DP}_{VH}(\mathcal{R}), \mathcal{R})$-chain.*

Example 5. Consider again the 3-CTRS $\mathcal{R}$ in Example 2 and the 2D-DPs in Example 4. There is an infinite $(\mathsf{DP}_V(\mathcal{R}), \mathsf{DP}_{VH}(\mathcal{R}), \mathcal{R})$-chain:

$$B \to_{\mathsf{DP}_{VH}(\mathcal{R})} F(a) \to^*_{\mathcal{R}} F(a) \to_{\mathsf{DP}_V(\mathcal{R})} G(a) \to^*_{\mathcal{R}} G(a) \to_{\mathsf{DP}_{VH}(\mathcal{R})} B$$

witnessing that $\mathcal{R}$ is *not* operationally terminating.

5 Mechanizing proofs of operational termination using 2D-DPs

In the following definition, we speak of $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, (\mathsf{ctrs}, \gamma))$-chains, for $\gamma = \mathsf{a}$ (resp. $\gamma = \mathsf{m}$) if arbitrary (resp. only minimal) chains are considered, in the sense of Definition 5; similarly, according to Remark 3, we speak of $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, (\mathsf{trs}, \gamma))$-chains if $\mathcal{P}$ and $\mathcal{R}$ are TRSs and $\mathcal{Q} = \emptyset$.

Definition 6 (CTRS problem). *A CTRS problem τ is a tuple $\tau = (\mathcal{P}, \mathcal{Q}, \mathcal{R}, e)$, where $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{R}$ are CTRSs, and $e \in \{\mathsf{ctrs}, \mathsf{trs}\} \times \{\mathsf{a}, \mathsf{m}\}$ is a flag. The CTRS problem τ is finite if there is no infinite minimal $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e)$-chain. The CTRS problem τ is infinite if $\mathcal{R}$ is non-operationally terminating or there is an infinite minimal $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e)$-chain.*

Definition 7 (CTRS processor). *A CTRS processor* Proc *is a mapping from CTRS problems into sets of CTRS problems. Alternatively, it can also return* "no"*. A CTRS processor* Proc *is*

- sound *if for all CTRS problems τ, we have that τ is finite whenever $\mathsf{Proc}(\tau) \neq \mathsf{no}$ and all CTRS problems in $\mathsf{Proc}(\tau)$ are finite.*
- complete *if for all CTRS problems τ, we have that τ is infinite whenever $\mathsf{Proc}(\tau) = \mathsf{no}$ or when $\mathsf{Proc}(\tau)$ contains an infinite CTRS problem.*

A (sound) processor transforms CTRS problems into (hopefully) *simpler* ones, in such a way that the existence of an infinite chain in the original CTRS problem implies the existence of an infinite chain in the transformed one. Here, 'simpler' usually means that fewer pairs are involved. Soundness is essential for proving *operational termination*; completeness for proving *non-operational termination*.

Processors are used in a *divide and conquer* scheme to incrementally simplify the original CTRS problem as much as possible, possibly decomposing it into (a tree of) smaller pieces which are independently treated in the same way. The trivial case comes when the set of pairs $\mathcal{P}$ becomes empty. Then, no infinite chain is possible, and the CTRS problem is finite. Such *positive* answer is propagated upwards in the decision tree. In some cases, a witness of an infinite chain is obtained; then a *negative* answer "no" can be provided and propagated upwards.

Theorem 3 (2D-DP-framework). *Let $\mathcal{R}$ be a CTRS. We construct* two *trees whose nodes are labeled with CTRS problems or* "yes" *or* "no"*, and whose roots are labeled with* $(\mathsf{DP}_H(\mathcal{R}), \emptyset, \mathcal{R}, (\mathsf{ctrs}, \gamma))$ *and* $(\mathsf{DP}_V(\mathcal{R}), \mathsf{DP}_{VH}(\mathcal{R}), \mathcal{R}, (\mathsf{ctrs}, \gamma))$*, respectively (for* $\gamma \in \{\mathsf{a}, \mathsf{m}\}$*). For every inner node labeled with* τ*, there is a sound processor* Proc *satisfying one of the following conditions:*

1. $\mathsf{Proc}(\tau) = \mathsf{no}$ *and the node has just one child that is labeled with* "no"*.*
2. $\mathsf{Proc}(\tau) = \emptyset$ *and the node has just one child that is labeled with* "yes"*.*
3. $\mathsf{Proc}(\tau) \neq \mathsf{no}$*,* $\mathsf{Proc}(\tau) \neq \emptyset$*, and the children of the node are labeled with the CTRS problems in* $\mathsf{Proc}(\tau)$*.*

If all leaves of both trees are labeled with "yes"*, then $\mathcal{R}$ is operationally terminating. If a leaf is labeled with* "no" *in some of the trees and all processors used on the path from the root to this leaf are complete, then $\mathcal{R}$ is operationally nonterminating.*

Our first processor *transfers* any proof (or disproof) of the finiteness of a 2D-DP problem to the standard DP framework for TRSs. In this way, all existing processors for the DP-framework are now available for the 2D-DP framework.

Theorem 4 (Shift to DP-Framework). *Let $\mathcal{P}$ and $\mathcal{R}$ be TRSs. Then,*

$$\mathsf{Proc}_{TRS}(\mathcal{P}, \emptyset, \mathcal{R}, (\mathsf{ctrs}, \gamma)) = \{(\mathcal{P}, \emptyset, \mathcal{R}, (\mathsf{trs}, \gamma))\}$$

is a sound and complete processor.

5.1 Dependency Graph

Given a CTRS problem $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e)$, we provide a notion of graph that is able to represent all infinite (*minimal*) chains of pairs as given in Definition 5.

Definition 8 (CTRS Graph of Pairs). *Let $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{R}$ be CTRSs. The* CTRS-graph $\mathsf{G}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e)$ *where $e = (\mathsf{ctrs}, \gamma)$ and $\gamma \in \{\mathsf{a}, \mathsf{m}\}$ has $\mathcal{P}$ as the set of nodes. Given $\alpha : u \to v$ if $c, \alpha' : u' \to v'$ if $c' \in \mathcal{P}$, there is an arc from α to α' if α, α' is a minimal $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e)$-chain for some substitution σ.*

In general, the CTRS graph is *not* computable due to the reachability conditions $\sigma(v)(\to_{\mathcal{R}}^{*} \circ \xrightarrow{\Lambda}_{\overline{\mathcal{Q}}})^{*}\sigma(u')$ (for $u \to v$ if $c \in \mathcal{P}$). Since the reachability problem for (conditional) rewriting is undecidable, we *approximate* it. Following [8], we approximate the CTRS-dependency graph as follows. Let $\mathrm{TCAP}_{\mathcal{R}}$ be:

$$\mathrm{TCAP}_{\mathcal{R}}(x) = y \text{ if } x \text{ is a variable, and}$$
$$\mathrm{TCAP}_{\mathcal{R}}(f(t_1, \ldots, t_k)) = \begin{cases} f([t_1], \ldots, [t_k]) & \text{if } f([t_1], \ldots, [t_k]) \text{ does not unify} \\ & \text{with } \ell \text{ for any } \ell \to r \text{ if } c \text{ in } \mathcal{R} \\ y & \text{otherwise} \end{cases}$$

where y is intended to be a new, fresh variable that has not yet been used and given a term s, $[s] = \mathrm{TCAP}_{\mathcal{R}}(s)$. We assume that ℓ shares no variable with $f([t_1], \ldots, [t_k])$ (rename if necessary). With $\mathrm{TCAP}_{\mathcal{R}}$ we approximate reachability problems as *unification*. According to Definitions 5 and 8, we have the following.

Definition 9 (Estimated connection). *Let $\mathcal{Q}$ and $\mathcal{R}$ be CTRSs, θ be a substitution, and $\alpha : u \to v$ if c to $\alpha' : u' \to v'$ if c' be two conditional rules. There is a* $(\mathcal{Q}, \mathcal{R}, \theta)$-connection *from α to α' if*

1. $\mathrm{TCAP}_{\mathcal{R}}(\theta(v))$ *and u' unify, or*
2. $\mathrm{TCAP}_{\mathcal{R}}(\theta(v))$ *and u'' unify with* mgu θ' *for some $\alpha'' : u'' \to v''$ if $c'' \in \mathcal{Q}$ and there is a $(\mathcal{Q} - \{\alpha''\}, \mathcal{R}, \theta')$-connection from α'' to α'.*

Definition 10 (Estimated CTRS Graph of Pairs). *Let $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{R}$ be CTRSs. The* estimated *CTRS-graph $\mathsf{EG}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e)$ has $\mathcal{P}$ as the set of nodes. There is an arc from α to α' if there is a $(\mathcal{Q}, \mathcal{R}, \epsilon)$-connection from α to α'.*

Remark 4. If $\mathcal{Q} = \emptyset$ and $\mathcal{P}, \mathcal{R}$ are TRSs, Definitions 8 and 10 specialize to the standard ones for TRSs [7, Definition 7] (and [8, Definition 12]).

Definition 11 (Graphs for a CTRS). *Let $\mathcal{R}$ be a CTRS. The* CTRS Dependency Graphs *(CTRSDGs) for $\mathcal{R}$ are $\mathsf{DG}_H(\mathcal{R}) = \mathsf{G}(\mathsf{DP}_H(\mathcal{R}), \emptyset, \mathcal{R}, e)$ and $\mathsf{DG}_V(\mathcal{R}) = \mathsf{G}(\mathsf{DP}_V(\mathcal{R}), \mathsf{DP}_{VH}(\mathcal{R}), \mathcal{R}, e)$*

The following processor decomposes a CTRS problem $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e)$ with graph $\mathsf{G}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e)$ according to the *strongly connected components* (SCCs) of the graph, i.e., cycles in $\mathsf{G}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e)$ that are not contained in any other cycle.

Theorem 5 (SCC processor). *Let $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{R}$ be CTRSs. Then,*

$$\mathsf{Proc}_{SCC}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e) = \{(\mathcal{P}'\mathcal{Q}, \mathcal{R}, e) \mid \mathcal{P}' \text{ are the pairs of an SCC in } \mathsf{G}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e)\}$$

is a sound and complete processor.

With Proc_{SCC}, we can *separately* work with the strongly connected components of $\mathsf{G}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e)$, disregarding other parts of the graph.

Example 6. For $\mathcal{R}$ in Example 3, $\mathsf{DG}_H(\mathcal{R})$ and $\mathsf{DG}_V(\mathcal{R})$ are shown in Figure 3. With $\tau_H = (\mathsf{DP}_H(\mathcal{R}), \emptyset, \mathcal{R})$, we have $\mathsf{Proc}_{SCC}(\tau_H) = \{\tau_{H1}, \tau_{H2}\}$, where $\tau_{H1} = (\{(17)\}, \emptyset, \mathcal{R})$ and $\tau_{H2} = (\{(18)\}, \emptyset, \mathcal{R})$. For $\tau_V = (\mathsf{DP}_V(\mathcal{R}), \mathsf{DP}_{VH}(\mathcal{R}), \mathcal{R})$ we get $\mathsf{Proc}_{SCC}(\tau_V) = \{\tau_{V1}\}$, where $\tau_{V1} = (\{(20)\}, \emptyset, \mathcal{R})$.

5.2 Use of orderings and argument filterings

The absence of infinite chains of pairs can be ensured by finding appropriate relations that are compatible with the rules and the pairs. In this way, we obtain *smaller* sets of pairs $\mathcal{P}' \subseteq \mathcal{P}$ by removing the *strict* pairs, i.e., those pairs $u \to v$ if $\bigwedge_{i=1}^{n} s'_i \to t'_i \in \mathcal{P}$ which are *compatible* with a well-founded relation $\sqsupset$. In the following, we provide precise definitions to achieve this.

Definition 12 (Conditional reduction triple). *Let $\mathcal{F}$ be a signature. A* conditional reduction triple $(\gtrsim, \succeq, \sqsupset)$ *consists of relations $\gtrsim, \succeq, \sqsupset$ on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $\gtrsim$ is a monotonic quasi-ordering and $\sqsupset$ is well-founded. Furthermore, we require $R \circ \sqsupset \subseteq \sqsupset$ or $\sqsupset \circ R \subseteq \sqsupset$ for $R \in \{\gtrsim, \succeq\}$ and $\gtrsim \circ \succeq \subseteq \gtrsim$ or $\gtrsim \circ \succeq \subseteq \succeq$.*

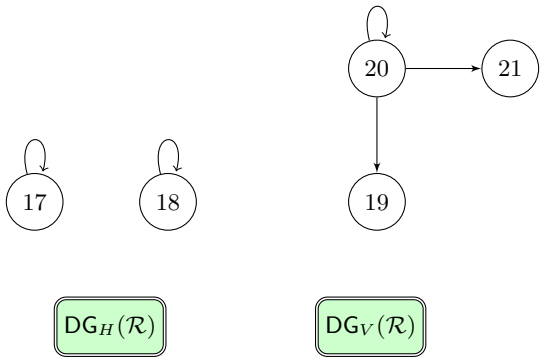


Fig. 3. Estimated DGs for $\mathcal{R}$ in Example 3

An *argument filtering* π for a signature $\mathcal{F}$ is a mapping that assigns to each k-ary function symbol $f \in \mathcal{F}$ an argument position $i \in \{1, \ldots, k\}$ or a (possibly empty) list $[i_1, \ldots, i_m]$ of argument positions with $1 \leq i_1 < \cdots < i_m \leq k$ [10]. The *trivial* argument filtering $\pi_\top(f) = [1, \ldots, k]$ (for each k-ary symbol $f \in \mathcal{F}$) is the argument filtering which does nothing. The signature $\mathcal{F}_\pi$ of symbols with filtered arguments consists of all function symbols $f \in \mathcal{F}$ such that $\pi(f)$ is some list $[i_1, \ldots, i_m]$, where, in $\mathcal{F}_\pi$, the arity of f is m. As usual, we give the same name to the version of $f \in \mathcal{F}$ that belongs to $\mathcal{F}_\pi$. An argument filtering π induces a mapping from $\mathcal{T}(\mathcal{F}, \mathcal{X})$ to $\mathcal{T}(\mathcal{F}_\pi, \mathcal{X})$, also denoted by π, which removes subterms:

$$\pi(t) = \begin{cases} t & \text{if } t \text{ is a variable} \\ \pi(t_i) & \text{if } t = f(t_1, \ldots, t_k) \text{ and } \pi(f) = i \\ f(\pi(t_{i_1}), \ldots, \pi(t_{i_m})) & \text{if } t = f(t_1, \ldots, t_k) \text{ and } \pi(f) = [i_1, \ldots, i_m] \end{cases}$$

Argument filterings provide a simple way to *remove* parts of the syntactic structure of a rule. In this way, we obtain simpler rules that are easier to compare.

Theorem 6 (Reduction triple processor). *Let $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$ be CTRSs, π be an argument filtering and $(\gtrsim, \succeq, \sqsupseteq)$ be a conditional reduction triple such that:*

1. *For all $\ell \to r$ if $\bigwedge_{i=1}^{n} s_i \to t_i \in \mathcal{R}$ and substitutions σ, if $\sigma(\pi(s_i)) \gtrsim \sigma(\pi(t_i))$ for all $1 \leq i \leq n$, then $\sigma(\pi(\ell)) \gtrsim \sigma(\pi(r))$.*
2. *For all $\overline{u} \to \overline{v}$ if $\bigwedge_{i=1}^{n} \overline{u}_i \to \overline{v}_i \in \mathcal{Q}$ and substitutions σ, if $\sigma(\pi(\overline{u}_i)) \gtrsim \sigma(\pi(\overline{v}_i))$ for all $1 \leq i \leq n$, then $\sigma(\pi(\overline{u})) \bowtie \sigma(\pi(\overline{v}))$ for some $\bowtie \in \{\gtrsim, \succeq, \sqsupseteq\}$.*
3. *For all $u \to v$ if $\bigwedge_{i=1}^{n} u_i \to v_i \in \mathcal{P}$ and substitutions σ, if $\sigma(\pi(u_i)) \gtrsim \sigma(\pi(v_i))$ for $1 \leq i \leq n$, then $\sigma(\pi(u)) \bowtie \sigma(\pi(v))$ for some $\bowtie \in \{\gtrsim, \succeq, \sqsupseteq\}$.*

Let $\mathcal{P}_\sqsupseteq$ ($\mathcal{Q}_\sqsupseteq$) be the set of rules $u \to v$ if $\bigwedge_{i=1}^{n} u_i \to v_i \in \mathcal{P}$ ($\overline{u} \to \overline{v}$ if $\bigwedge_{i=1}^{n} \overline{u}_i \to \overline{v}_i \in \mathcal{Q}$) satisfying that for all substitutions σ, $\sigma(\pi(u)) \sqsupseteq \sigma(\pi(v))$ (resp.

$\sigma(\pi(\overline{u})) \sqsupset \sigma(\pi(\overline{v}))$*) holds whenever* $\sigma(\pi(u_i)) \gtrsim \sigma(\pi(v_i))$ *(resp.* $\sigma(\pi(\overline{u}_i)) \gtrsim \sigma(\pi(\overline{v}_i))$*) holds for all* $1 \le i \le n$*. Then,*

$$\mathsf{Proc}_{RT}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e) = \{(\mathcal{P} - \mathcal{P}_{\sqsupset}, \mathcal{Q} - \mathcal{Q}_{\sqsupset}, \mathcal{R}, e)\}$$

is a sound and complete processor.

Example 7. Consider the CTRS $\mathcal{R}$ in Example 3 and the CTRS problems τ_{H1}, τ_{H2} and τ_{V1} in Example 6. We apply Proc_{RT} to those problems using the same reduction triple $(\ge, \ge, >)$ induced by the polynomial interpretation

$$\begin{array}{cccc} [\mathsf{false}] = 0 & [\mathsf{true}] = 0 & [\mathsf{0}] = 0 & [\mathsf{s}](x) = x + 1 \\ [\mathsf{less}](x) = 0 & [\mathsf{minus}](x, y) = x & [\mathsf{pair}](x, y) = 0 & [\mathsf{quotrem}](x, y) = 0 \\ [\mathsf{LESS}](x, y) = x & [\mathsf{MINUS}](x, y) = x & [\mathsf{QUOTREM}](x, y) = x & \end{array}$$

by $s \ge t$ if $[s] \ge [t]$ and $s > t$ if $[s] > [t]$. We have:

$$\begin{array}{rcl} [\mathsf{less}(x, \mathsf{0})] = 0 & \ge & 0 = [\mathsf{false}] \\ [\mathsf{less}(\mathsf{0}, s(x))] = 0 & \ge & 0 = [\mathsf{true}] \\ [\mathsf{less}(s(x), s(y))] = 0 & \ge & 0 = [\mathsf{less}(x, y)] \\ [\mathsf{minus}(\mathsf{0}, s(y))] = 0 & \ge & 0 = [\mathsf{0}] \\ [\mathsf{minus}(x, \mathsf{0})] = x & \ge & x = [x] \\ [\mathsf{minus}(s(x), s(y))] = x + 1 & \ge & x = [\mathsf{minus}(x, y)] \\ [\mathsf{quotrem}(\mathsf{0}, s(y))] = 0 & \ge & 0 = [\mathsf{pair}(\mathsf{0}, \mathsf{0})] \\ [\mathsf{quotrem}(s(x), s(y))] = 0 & \ge & 0 = [\mathsf{pair}(\mathsf{0}, s(x))] \\ [\mathsf{quotrem}(s(x), s(y))] = 0 & \ge & 0 = [\mathsf{pair}(s(q), r)] \\ [\mathsf{LESS}(s(x), s(y))] = x + 1 & > & x = [\mathsf{LESS}(x, y)] \\ [\mathsf{MINUS}(s(x), s(y))] = x + 1 & > & x = [\mathsf{MINUS}(x, y)] \\ [\mathsf{QUOTREM}(s(x), s(y))] = x + 1 & > & x = [\mathsf{QUOTREM}(\mathsf{minus}(x, y), s(y))] \end{array}$$

Since we do not use the conditions to prove the compatibility of the ordering with the rules, we omit them in the example. This proves proves τ_{H1}, τ_{H2} and τ_{V1} finite. Thus, $\mathcal{R}$ is operationally terminating.

Theorem 7 (Unsatisfiable rules). *Let* $\mathcal{P}$*,* $\mathcal{Q}$*, and* $\mathcal{R}$ *be CTRSs,* π *be an argument filtering and* $(\gtrsim, \succeq, \sqsupset)$ *be a conditional reduction triple such that:*

1. *For all* $\ell \to r$ if $\bigwedge_{i=1}^{n} s_i \to t_i \in \mathcal{R}$ *and substitutions* σ*, if* $\sigma(\pi(s_i)) \gtrsim \sigma(\pi(t_i))$ *for all* $1 \le i \le n$*, then* $\sigma(\pi(\ell)) \gtrsim \sigma(\pi(r))$*.*
2. *For all* $\overline{u} \to \overline{v}$ if $\bigwedge_{i=1}^{n} \overline{u}_i \to \overline{v}_i \in \mathcal{Q}$ *and substitutions* σ*, if* $\sigma(\pi(\overline{u}_i)) \gtrsim \sigma(\pi(\overline{v}_i))$ *for all* $1 \le i \le n$*, then* $\sigma(\pi(\overline{u})) \bowtie \sigma(\pi(\overline{v}))$ *for some* $\bowtie \in \{\gtrsim, \succeq, \sqsupset\}$*.*
3. *For all* $u \to v$ if $\bigwedge_{i=1}^{n} u_i \to v_i \in \mathcal{P}$ *and substitutions* σ*, if* $\sigma(\pi(u_i)) \gtrsim \sigma(\pi(v_i))$ *for all* $1 \le i \le n$*, then* $\sigma(\pi(u)) \bowtie \sigma(\pi(v))$ *for some* $\bowtie \in \{\gtrsim, \succeq, \sqsupset\}$*.*

Let $\mathcal{P}_{\sqsupset}$ *(*$\mathcal{Q}_{\sqsupset}$*,* $\mathcal{R}_{\sqsupset}$*) be the set of rules* $u \to v$ if $\bigwedge_{i=1}^{n_{\mathcal{P}}} u_i \to v_i \in \mathcal{P}$ *(*$\overline{u} \to \overline{v}$ if $\bigwedge_{j=1}^{n_{\mathcal{Q}}} \overline{u}_j \to \overline{v}_j \in \mathcal{Q}$*,* $\ell \to r$ if $\bigwedge_{k=1}^{n_{\mathcal{R}}} s_k \to t_k \in \mathcal{R}$*) satisfying that for all*

substitution σ, there is i, $1 \le i \le n_{\mathcal{P}}$ (j, $1 \le j \le n_{\mathcal{Q}}$, k, $1 \le k \le n_{\mathcal{R}}$) such that $\sigma(\pi(v_i)) \sqsupseteq \sigma(\pi(u_i))$ (resp. $\sigma(\pi(\overline{v}_j)) \sqsupseteq \sigma(\pi(\overline{u}_j))$, $\sigma(\pi(t_k)) \sqsupseteq \sigma(\pi(s_k))$) Then,

$$\mathsf{Proc}_{UR}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, e) = \{(\mathcal{P} - \mathcal{P}_{\sqsupset}, \mathcal{Q} - \mathcal{Q}_{\sqsupset}, \mathcal{R} - \mathcal{R}_{\sqsupset}, e)\}$$

is a sound and (if $\mathcal{R}_{\sqsupset} = \emptyset$ or $e = (\rho, \mathsf{a})$) complete processor.

Example 8. For $\mathcal{R}$ in Example 1, $\mathsf{DP}_H(\mathcal{R}) = \{F(k(a), k(b), x) \to F(x, x, x)\}$, and $\mathsf{DP}_V(\mathcal{R}) = \mathsf{DP}_{VH}(\mathcal{R}) = \emptyset$. For the initial CTRS problem $(\mathsf{DP}_H(\mathcal{R}), \emptyset, \mathcal{R}, (\mathsf{ctrs}, \mathsf{m}))$, the following interpretation:

$$\begin{array}{rrrrr} [a] = 0 & [b] = 0 & [c](x) = 0 & [d] = 1 & [f](x, y, z) = 0 \\ [g](x) = 0 & [h](x) = 0 & [k](x) = 0 & [F](x, y, z) = 0 & \end{array}$$

can be used to generate a triple $(\geq, \geq, >)$ which can be used to prove $\mathcal{R}$ operationally terminating. Since $[h(x)] = 0$ and $[d] = 1$, we have $[d] > [h(x)]$. With Proc_{UR}, we can remove rule (4) from $\mathcal{R}$. The new CTRS problem $(\mathsf{DP}_H(\mathcal{R}), \emptyset, \mathcal{R}')\}, (\mathsf{ctrs}, \mathsf{m}))$, where $\mathcal{R}' = \mathcal{R} - \{(4)\}$, satisfies the conditions in Theorem 4 for a shift to a DP problem $\tau_{\mathsf{trs}} = (\mathsf{DP}_H(\mathcal{R}), \emptyset, \mathcal{R}', (\mathsf{trs}, \mathsf{m}))$ that can then be solved by using any processor for TRSs. For instance, the forward instantiation processor [7, Definition 28] can be used to prove finiteness of τ_{trs}.

6 Related work and conclusions

To the best of our knowledge, this is the first correct and complete characterization of operational termination of deterministic 3-CTRS which is based on the notion of dependency pair. The notion of minimal operationally nonterminating term and the properties explored here (Section 3) are also new in the literature.

The recent *Conditional Dependency Pairs* (CDPs) by Nakamura et al. [12] apply to a restricted subclass of *1-CTRSs*: the condition c in the 1-rules ($\ell \to r$ if c) considered in [12] is a *term* rather than a sequence $s_1 \to t_1, \ldots, s_n \to t_n$. An instance $\sigma(c)$ of condition c is satisfied if and only if $\sigma(c) \to^* \mathsf{true}$. For the 1-CTRSs considered in [12], our proposal generates a *subset* of the pairs considered in [12, Definition 3.1], i.e., $\mathsf{DP}_H(\mathcal{R}) \cup \mathsf{DP}_V(\mathcal{R}) \cup \mathsf{DP}_{VH}(\mathcal{R}) \subseteq CDP(\mathcal{R})$. Often, the inclusion is strict due to our more restrictive generation of pairs. Their notion of chain ([12, Definition 3.2]) is also different to our Definition 5.

As remarked in the introduction, existing tools for proving termination of conditional TRSs currently use transformation techniques. We are not aware of any implementation of *direct* methods. The transformation which is typically used for this purpose is $\mathcal{U}$ in [13, Definition 7.2.48]. This transformation is not complete, however. For instance, $\mathcal{U}(\mathcal{R})$ is not terminating for $\mathcal{R}$ in Example 1, but we proved that $\mathcal{R}$ is operationally terminating in Example 8. Furthermore, when $\mathcal{U}(\mathcal{R})$ is terminating, tools may fail to find a proof. This is often due to the loss of information introduced by transformations, and also to the presence of new symbols and rules that prevent the search process from finding a proof. The techniques presented in this paper have been included as part of the tool

MU-TERM[3] [2]. The first benchmarks of existing examples in the literature are very positive and show that the 2D-DP framework permits simple and fast proofs like the ones in the examples of this paper. This will also make possible to have these techniques available to tools like MTT [4], which use MU-TERM as a backend for achieving proofs of operational termination of more general theories like membership equational programs or order-sorted rewrite theories. This will require the extension of the techniques presented here to the case of order-sorted conditional rewrite theories with types and subtypes, and where rewriting can take place modulo axioms B. This is envisaged as an interesting subject for future work.

References

1. T. Arts and J. Giesl. Termination of Term Rewriting Using Dependency Pairs. *Theoretical Computer Science*, 236(1–2):133–178, 2000.
2. B. Alarcón, R. Gutiérrez, S. Lucas, R. Navarro-Marset. Proving Termination Properties with MU-TERM. In *Proc. of AMAST'10*, LNCS 6486:201-208, 2011.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. All About Maude – A High-Performance Logical Framework. LNCS 4350, Springer-Verlag, 2007.
4. F. Durán, S. Lucas, and J. Meseguer. MTT: The Maude Termination Tool (System Description). *Proc. of IJCAR'08*. LNAI 5195:313–319, Springer-Verlag, 2008.
5. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
6. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *Proc. of LPAR'04*, LNAI 3452:301–331, Springer-Verlag, 2004.
7. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automatic Reasoning*, 37(3):155–203, 2006.
8. J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and Disproving Termination of Higher-Order Functions. In *Proc. of FroCoS'05*, LNAI 3717:216-231, Springer-Verlag, 2005.
9. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Proc. of IJCAR'06*, LNAI 4130:281–286, Springer-Verlag, 2006.
10. K. Kusakari, M. Nakamura, and Y. Toyama. Argument Filtering Transformation. In *Proc. of PPDP'99*, LNCS 1702:47–61, Springer-Verlag, 1999.
11. S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95:446–453, 2005.
12. M. Nakamura, K. Ogata, and K. Futatsugi. On Proving Operational Termination Incrementally with Modular Conditional Dependency Pairs. International Journal of Computer Science, 40:2, 2013.
13. E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, Apr. 2002.
14. F. Schernhammer and B. Gramlich. Characterizing and proving operational termination of deterministic conditional term rewriting systems. *Journal of Logic and Algebraic Programming* 79:659-688, 2010.
15. F. Schernhammer and B. Gramlich. VMTL - A Modular Termination Laboratory. In Proc. of RTA'09. LNCS 5595:285–294, Springer-Verlag, 2009.

[3] We thank Raúl Gutiérrez for the implementation of the 2D-DP Framework.

FunKons: Component-Based Semantics in K

Peter D. Mosses and Ferdinand Vesely

Swansea University, Swansea, SA2 8PP, United Kingdom
p.d.mosses@swansea.ac.uk, csfvesely@swansea.ac.uk

Abstract. Modularity has been recognised as a problematic issue of programming language semantics, and various semantic frameworks have been designed with it in mind. Reusability is another desirable feature which, although not the same as modularity, can be enabled by it. The K Framework, based on Rewriting Logic, has good modularity support, but reuse of specifications is not as well developed.
The PLanCompS project is developing a framework providing an open-ended collection of reusable components for semantic specification. Each component specifies a single fundamental programming construct, or 'funcon'. The semantics of concrete programming language constructs is given by translating them to combinations of funcons. In this paper, we show how this component-based approach can be seamlessly integrated with the K Framework. We give a component-based definition of CinK (a small subset of C++), using K to define its translation to funcons as well as the (dynamic) semantics of the funcons themselves.

1 Introduction

Even very different programming languages often share similar constructs. Consider OCaml's conditional '**if** E_1 **then** E_2 **else** E_3' and the conditional operator 'E_1 **?** E_2 **:** E_3' in C. These constructs have different concrete syntax but similar semantics, with some variation in details. We would like to exploit this similarity when defining formal semantics for both languages by reusing parts of the OCaml specification in the C specification. With traditional approaches to semantics, reuse through 'copy-paste-and-edit' is usually the only option that is available to us. By default, this is also the case with the K Framework [9,13]. This style of specification reuse is not systematic, and prone to error.

The semantic framework currently being developed by the PLanCompS project[1] provides *fundamental constructs* (funcons) that address the issues of reusability in a systematic manner. Funcons are small semantic entities which express essential concepts of programming languages. These formally specified components can be composed to capture the semantics of concrete programming language constructs. A specification of Caml Light has been developed as an initial case study [3] and a case study on C# is in progress.

[1] http://www.plancomps.org/

For example, the funcon `if-true` can be used to specify OCaml's conditional expression. Semantics is given by defining a translation from the concrete construct to the corresponding funcon term:

$$[\![\texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3]\!] = \texttt{if-true}([\![E_1]\!], [\![E_2]\!], [\![E_3]\!])$$

Since the conditional operator in C uses integer valued expressions as the condition, its translation will reflect this:

$$[\![E_1\ \texttt{?}\ E_2\ \texttt{:}\ E_3]\!] = \texttt{if-true}(\texttt{not}(\texttt{equal}([\![E_1]\!], 0)), [\![E_2]\!], [\![E_3]\!])$$

We could also define an `if-non-zero` funcon that would match the C-conditional semantics exactly. However, the translation using `if-true` is so simple that there wouldn't be much advantage in doing so. We can reuse the `if-true` funcon, and with it, its semantic definition. This way, we also make the difference between the OCaml and C conditional construct transparent. Section 2 provides more information on funcons.

PLanCompS uses MSOS [10], a modular variant of structural operational semantics [11], to formally define individual funcons. However, the funcon approach can be seamlessly integrated with other sufficiently modular specification frameworks. We have tested the use of funcons with the K Framework by giving a specification of CinK [8,9], a pedagogical subset of C++. We have defined both the translation of CinK to funcons and the semantics of the funcons using K's rewrite rules. The complete prototyped specification is available online.[2] Also included are the CinK test programs which we have used to test our specification. Interested readers may run these programs themselves using the K tool.

In this paper, we present our specification of the CinK translation (Section 3) and illustrate the definition of the semantics of funcons involved in it (Section 4). Section 5 offers an overview of related work and alternative approaches. We conclude and suggest directions of future work in Section 6.

2 Fundamental Constructs

As mentioned in the Introduction, the PLanCompS project is developing an open-ended collection of fundamental programming constructs, or 'funcons'. Many funcons correspond closely to simplified programming language constructs. However, each funcon has *fixed* syntax and semantics. For example, the funcon written `assign(`*`E1`*`,`*`E2`*`)` has the effect of evaluating *`E1`* to a variable, *`E2`* to a value (in any order), then assigning the value to the variable; it is well-typed only if *`E1`* is of type `variables(`*`T`*`)` and *`E2`* is of type *`T`*. In contrast, the language construct written '*`E1`* `=` *`E2`*' may be interpreted as an assignment or as an equality test (and its well-typedness changes accordingly) depending on the language.

The syntax or *signature* of a funcon determines its name, how many arguments it takes (if any), the sort of each argument, and the sort of the result. The following *computation sorts* reflect fundamental conceptual and semantic distinctions in programming languages.

[2] `http://cs.swan.ac.uk/~csfvesely/wrla2014/`

- The sort `Comm` (commands) is for funcons (such as `assign(E1,E2)`) that are executed only for their effects; on normal termination, a command computes the fixed value `skip`.
- The sort `Expr` (expressions) is for funcons (such as `stored-value(E)` and `bound-value(I)`) that compute values of sort `Values`.
- The sort `Decl` (declarations) is for funcons (such as `bind-value(I,E)`) that compute environments of sort `Environments`, which represent sets of bindings between identifiers and values.

All computation sorts include their sorts of computed values as subsorts: a value takes no steps at all to compute itself.

One of the aims of the PLanCompS project is to establish an online repository of funcons (and data types) for anybody to use 'off-the-shelf' as components of language specifications. The project is currently testing the reusability of existing funcons and developing new ones in connection with some major case studies (including Caml Light, C#, and Java). Because individual funcons are meant to represent *fundamental* concepts in programming languages, many funcons (expressing, e.g., sequencing, conditionals, variable lookup and dereferencing) have a high potential for reuse. In fact, many funcons used in the Caml Light case study appear in the semantics of CinK presented in the following section.

The nomenclature and notation for the existing funcons are still evolving, and they will be finalised only when the case studies have been completed, in connection with the publication of the repository. Observant readers are likely to notice some (minor) differences between the funcon names used in this paper and in previous papers (e.g. [3]).

Regardless of the details of funcon notation, funcons can be algebraically composed to form funcon terms, according to their argument sorts (strictly lifted to corresponding computation sorts) and result sorts. Well-*formedness* of funcon terms is context-free: `assign(E1,E2)` is a well-formed funcon term whenever `E1` and `E2` are well-formed funcon terms of sort `Expr`. In contrast, well-*typedness* of funcon terms is generally context-sensitive. For example, the funcon term `assign(bound-value(I),42)` is well-typed only in the scope of a declaration that binds `I` to an integer variable. Dynamic semantics is defined for all well-formed terms; execution of ill-typed terms usually fails.

The composability of funcons does *not* depend on features such as whether they might have side effects, terminate abruptly, diverge, spawn processes, interact, etc. This is crucial for the reusability of the funcons. The semantics of each funcon has to be specified without regard to the context in which it might be used, which requires a highly modular specification framework. Funcon specifications have previously been given in MSOS, Rewriting Logic, ASF+SDF, and action notation. Here, we explore specifying funcons in K, following Roşu.[3]

A component-based semantics of a programming language is specified by a context-free grammar for an abstract syntax for the language, together with a family of inductively specified functions translating abstract syntax trees to

[3] `k/examples/funcons` in the stable K distribution at `http://www.kframework.org`

funcon terms. The static and dynamic semantics of a program is given by that of the resulting funcon term. As mentioned above, funcons have fixed syntax and semantics. Thus, evolution of a language is expressed as changes to translation functions. If the syntax or semantics of the programming language changes, the definition of the translation function has to be updated to reflect this.

Tool support for translating programs to funcon terms, and for executing the static and dynamic semantics of such terms, has previously been developed in Prolog [2], Maude [1] and ASF+SDF. We now present our experiment with K, focusing on dynamic semantics.

3 CinF: a Funcon Specification of CinK

This section presents an overview of our CinK specification using funcons. We include examples from the K sources of the specification. A selection of definitions of funcons involved in the specification can be found in Section 4.

CinK is a pedagogical subset of C++ [8,9] used for experimentation with the K Framework. The original report [8] presents the language in seven iterations. The first specifies a basic imperative language; subsequent iterations extend it with threads, model-checking, references, pointers, and unidimensional and multi-dimensional arrays. Our specification starts with only an expression language which we extend with declarations, statements, functions, threads, references, and pointers. The extensions follow the order of the CinK iterations; however, we omit arrays and support for model-checking.

The grammar which we have used for our specification is a simplified grammar matching CinK derived from the C++ grammar found in the standard [7, Appendix A].

We invite the reader to compare our specification by translation to funcons with the original K specification of CinK in [8]. Our hope is that our translation functions, together with the suggestive naming of funcons, give a rough understanding of the semantics of language constructs, even before looking at the semantics of funcons themselves.

3.1 Simple Expressions

To give semantics for expressions we use the translation function `evaluate⟦_⟧ : Expression → Expr`. It produces a funcon term (of sort `Expr`) which, when executed, evaluates the argument expression.

Definitions for arithmetic expressions in CinK can be given very straightforwardly using data operations, which all extend to strict funcons on `Expr`. For example, semantics of the multiplication operator is expressed as the application of the operation `int-times` to translations of operand expressions (numeric types in CinK are limited to integers with some common operations):

```
rule evaluate⟦ E1:Expression * E2:Expression ⟧ ⇒
    int-times(evaluate⟦ E1 ⟧, evaluate⟦ E2 ⟧)
```

The 'short-circuit and' operator can be readily expressed using a conditional funcon, which is strict only in its first argument. The (obvious) K definition for `if-true` can be found in Section 4.

```
rule evaluate⟦ E1:Expression && E2:Expression ⟧ ⇒
    if-true(evaluate⟦ E1 ⟧, evaluate⟦ E2 ⟧, false)
```

We will use the generic `if-true` funcon later in this section to define the conditional statement.

3.2 Variables, Blocks and Scope

Bindings and Variables Semantics of declarations are given using the translation function `elaborate⟦_⟧ : DeclarationSeq → Decl`. The `bind-value(I,V)` funcon binds the identifier `I` to the value `V`, producing a 'small' environment containing only the newly created binding. To allocate a new variable of a specified type we use `allocate`. In Caml Light, `bind-value` was used for individual name-value bindings in let-expressions, and `allocate` for reference data types (e.g. '`ref int`').

```
rule elaborate⟦ T:TypeSpecifier I:Id ; ⟧ ⇒
    bind-value(I, allocate(variables(type⟦ T ⟧)))
```

In relation to variables, CinK (following C++) distinguishes between two general categories of expressions: *lvalue*- and *rvalue*-expressions. We express this distinction by having different translation functions for expressions in lvalue and rvalue contexts: in addition to `evaluate⟦_⟧`, we define `evaluate-lval⟦_⟧` and `evaluate-rval⟦_⟧`. The default function `evaluate⟦_⟧` produces terms evaluating lvalue and rvalue expressions according to their category. When an expression is expected to evaluate to an *lvalue*, we use `evaluate-lval⟦_⟧`, which is undefined on rvalue expressions. When an *rvalue* is expected, we use `evaluate-rval⟦_⟧` which produces terms evaluating all expressions into rvalues. For lvalue expressions it returns the corresponding stored value, i.e., it serves as an lvalue-to-rvalue conversion.

The addition of variables also affects our translations of simple expressions and we need to update them. For example, numeric operations expect an rvalue and thus the operands are now translated using `evaluate-rval⟦_⟧`.

To get the variable bound to an identifier in the current environment we use `bound-value`. A variable is dereferenced using `stored-value`. The semantics for an identifier appearing in an lvalue or rvalue context is thus:

```
rule evaluate-lval⟦ I:Id ⟧ ⇒ bound-value(I)
rule evaluate-rval⟦ I:Id ⟧ ⇒ stored-value(evaluate-lval⟦ I ⟧)
```

Blocks and Controlling Scope We distinguish between declaration statements and other statements within a block using funcons `scope` and `seq`. The funcon `scope(D,X)` evaluates `X` in the current environment overridden with the environment computed by `D`. A declaration statement within a block produces a new environment that is valid until the end of the block:

```
rule execute[[ BD:BlockDeclaration SS:StatementSeq ]] ⇒
    scope(elaborate[[ BD ]], execute[[ SS ]])
```

The function `execute[[_]] : StatementSeq → Comm` translates statements to funcon commands.

For all other kinds of statements in a block we use the simple sequencing funcon `seq(C,X)` which executes the command `C` for side effects, then executes `X`.

```
rule execute[[ BS:BlockStatement SS:StatementSeq ]] ⇒
    seq(execute[[ BS ]], execute[[ SS ]])
```

To accumulate multiple declarations into one environment we use the `accum` funcon. The funcon `accum(D1,D2)` is similar to `scope`, except its result is the environment produced by elaborating declaration `D2` and overriding the environment computed by `D1` with it. This matches the semantics of a multi-variable declaration:

```
rule elaborate[[ T:TypeSpecifier ID:InitDeclarator ,
                                 IDL:InitDeclaratorList ; ]] ⇒
    accum(elaborate[[ T ID ; ]], elaborate[[ T IDL ; ]])
```

Note that `accum` is strict in its first argument, so the correct order of evaluation is enforced.

Although Caml Light and CinK are quite different languages, all the funcons we needed so far for CinK here are reused from [3].

3.3 Assignment and Control Statements

The basic construct for updating variables in CinK/C++ is the assignment expression '`E1 = E2`', where the expression `E1` is expected to evaluate to an lvalue, to which the rvalue of `E2` will be assigned. The value of the whole expression is the lvalue of `E1`. Semantics of assignment is a rather simple translation using the `assign-giving-variable` funcon:

```
rule evaluate-lval[[ E1:Expression = E2:Expression ]] ⇒
    assign-giving-variable(evaluate-lval[[ E1 ]], evaluate-rval[[ E2 ]])
```

The funcon `assign-giving-variable` is strict in both arguments but not sequentially, so the arguments are evaluated in an unspecified order. The funcon assigns the value given as its second argument to the variable given as its first argument and returns this variable as result.

CinK has boolean-valued conditions and the translations of while- and if-statements are trivial:

```
rule execute[[ while ( E:Expression ) S:Statement ]] ⇒
    while-true(evaluate-rval[[ E ]], execute[[ S ]])
rule execute[[ if ( E:Expression ) S:Statement ]] ⇒
    execute[[ if ( E ) S else { } ]]
rule execute[[ if ( E:Expression ) S1:Statement else S2:Statement ]] ⇒
    if-true(evaluate-rval[[ E ]], execute[[ S1 ]], execute[[ S2 ]])
```

3.4 Function Definition and Calling

We represent functions as *abstraction values* which wrap any computation as a value. An abstraction can be passed as a parameter, bound to an identifier, or stored like any other value. To turn a funcon term into an abstraction, we use the `abstraction` value constructor. The funcon `apply` applies an abstraction to a value and the abstraction may refer to the passed value using `given`. Multiple parameters can be passed as a tuple constructed via tuple value constructors.

A function call expression simply applies the abstraction to translated arguments:

```
rule evaluate-rval⟦ E1:Expression ( E2:Expression ) ⟧ ⇒
    apply(evaluate-rval⟦ E1 ⟧, evaluate-params⟦ tuple(E2) ⟧)
```

At this stage the language only supports call-by-value semantics and so each parameter is evaluated to an *rvalue* before being passed to a function. The translation function `evaluate-params⟦_⟧` (defined in terms of `evaluate-rval⟦_⟧`) recurses through the parameter expressions and constructs a tuple.

```
rule evaluate-params⟦ tuple(E1:Expression , E2:Expression) ⟧ ⇒
    tuple-prefix(evaluate-rval⟦ E1 ⟧, evaluate-params⟦ tuple(E2) ⟧)
rule evaluate-params⟦ tuple(E:Expression) ⟧ ⇒
    tuple-prefix(evaluate-rval⟦ E ⟧, tuple(.))
```

We have introduced the auxiliary abstract syntax `tuple(E)` to ensure that parameters separated by commas are not interpreted as a comma-operator expression.

We use *patterns* as translations of function parameters. Patterns themselves are abstractions which compute an environment when applied to a matching value. The pattern for passing a single parameter by value allocates a variable of the corresponding type and binds it to an identifier; then it assigns the parameter value to the variable and returns the resulting environment.

```
rule pattern⟦ T:TypeSpecifier I:Id ⟧ ⇒
    abstraction(
      accum(bind-value(I, allocate(variables(type⟦ T ⟧))),
        decl-effect(assign(bound-value(I), given))))
```

Here we use the funcon `decl-effect(C)`, which allows using a command `C` as a declaration. It is an abbreviation for `seq(C,bindings(.))`.

Roughly, the semantics of a function definition is to allocate storage for an abstraction of the corresponding type, bind it to the function name, and use it to store an abstraction of the function body. Looking closer, the definition has to deal with some more details:

```
rule elaborate⟦ T:TypeSpecifier I:Id ( PDL:ParameterDeclarationList )
                CS:CompoundStatement ⟧ ⇒
    decl-effect(assign(bound-value(I),
      close(abstraction(
        scope(match-compound(pattern-tuple⟦ PDL ⟧, given),
          catch(seq(execute⟦ CS ⟧, throw(variant(returned, null))),
            abstraction(original(returned, given))))))))
```

Within the abstraction we use `match-compound` to match the passed value against the pattern tuple constructed from individual parameter patterns. The translation of the function body is evaluated in the environment produced by this matching (`scope`). Since a return statement abruptly terminates a function returning a value, we represent return statements as exceptions containing a value tagged with the atom '`returned`' and wrap the function body in a handler. The `catch` funcon catches the exception and the handling abstraction retrieves the value tagged with '`returned`', making it the return value of the whole function. In case there was no return statement in the body of the function, we throw a '`returned`' with `null`. Using `close` we form a closure of the abstraction with respect to the definition-time environment. This imposes static scopes for bindings.

As mentioned above, an explicit return statement translates to throwing a value tagged with '`returned`'. A parameterless return throws a `null`.

```
rule execute⟦ return E:Expression; ⟧ ⇒
    throw(variant(returned, evaluate-rval⟦ E ⟧))
rule execute⟦ return ; ⟧ ⇒ throw(variant(returned, null))
```

As a simple way of allowing self- and mutually recursive function definitions, we pre-allocate function variables and bind all function names declared at the top-level in a global environment using `evaluate-forwards⟦_⟧`. Then we combine this environment with the elaboration of full function definitions and other declarations. The `main` function is called in the scope of the global environment.

```
rule translate⟦ DS:DeclarationSeq ⟧ ⇒
    scope(accum(elaborate-forwards⟦ DS ⟧, elaborate⟦ DS ⟧),
      effect(apply(evaluate-rval⟦ main ⟧, tuple(.))))
```

Because function identifiers are already bound when the full function definition is elaborated, the full definition only assigns the abstraction to the pre-allocated variable.

3.5 Threads

The second iteration in the original CinK report adds very basic thread support to the language. Spawning a thread in CinK mimics the syntax of using the `std::thread` class from the C++ standard library. However, instead of referring to the standard library, semantics is given to the construct directly.

```
rule elaborate⟦ std::thread I1:Id ( I2:Id , E:Expression ) ; ⟧ ⇒
    decl-effect(effect(spawn(close(abstraction(evaluate⟦ I2 (E) ⟧)))))
```

The funcon `spawn(A)` creates a new thread in which the abstraction `A` will be applied. In our case the abstraction contains a function call corresponding to the parameters given to the thread constructor.

3.6 References

A reference in C++ is an alias for a variable, i.e., it introduces a new name for an already existing variable.

```
rule elaborate[[ T:TypeSpecifier & I:Id = E:Expression ]] ⇒
    bind-value(I, evaluate-lval[[ E ]])
```

The expression `E` is expected to be an lvalue and we bind the resulting variable to identifier `I`. We are assuming that the input program is statically correct and thus the variable will have the right type.

A reference parameter pattern simply binds `I` to the given variable.

```
rule pattern[[ T:TypeSpecifier & I:Id ]] ⇒
    abstraction(bind-value(I, given))
```

Before introducing references, we evaluated function parameters to an rvalue. Now the function `evaluate-param[[_]]` has to be *redefined* in terms of `evaluate[[_]]` instead of `evaluate-rval[[_]]`. Dereferencing is handled conditionally inside the parameter pattern.

```
rule pattern[[ T:TypeSpecifier I:Id ]] ⇒
    abstraction(
      accum(bind-value(I, allocate(variables(type[[ T ]]))),
        decl-effect(assign(bound-value(I), current-value(given)))))
```

The funcon `current-value` dereferences its parameter if it is a variable (lvalue), otherwise returns the parameter itself.

3.7 Pointers

The last CinK extension that we consider is the addition of pointers to the language. Pointers are variables that hold addresses to other objects in memory. A pointer declaration allocates a new object for holding locations (variables in our terminology). Our semantics of declarations uses types to allocate storage and a pointer declaration complicates matters. Here we present a simplified version only supporting single-level indirection. The complete version has to deal with the notoriously complicated syntax for pointer declarations in C++.

```
rule elaborate[[ T:TypeSpecifier * I:Id ; ]] ⇒
    accum(bind-value(I, allocate(variables(type[[ T * ]]))))
rule type[[ T:TypeSpecifier * ]] ⇒ pointers(type[[ T ]])
```

For our full specification of pointers, we refer the reader to the online material.

Explicit dereferencing of a pointer variable amounts to retrieving the value stored in the pointer. This value is the location to which the pointer is pointing. This is expressed in our translation:

```
rule evaluate-lval[[ * E:Expression ]] ⇒
    stored-value(evaluate-rval[[ E ]])
```

A Note on Reuse The complete funcon definition of CinK available online uses 26 funcons. Of these, 19 have been previously used in the specification of Caml Light and only 7 were introduced in the present work, 3 of which are just abbreviations for longer funcon terms. It is thus possible to conclude that the degree of reuse of funcons between the Caml Light and CinK specifications is high, even if the languages are quite different.

3.8 Configuration

The configuration of the final iteration of our specification is as follows:

```
configuration
    <T>
      <threads>
        <thread multiplicity="*">
          <name> main:Threads </name>
          <k> translate[[ $PGM:TranslationUnit ]] </k>
          <xstack> .List </xstack>
          <context>
            <env> .Map </env>
            <given> no-value </given>
          </context>
        </thread>
      </threads>
      <store> .Map </store>
      <output stream="stdout"> .List </output>
      <input stream="stdin"> .List </input>
    </T>
```

It appears that this configuration could be generated from the K rules defining the funcons used in our specification of CinK. It is unclear to us whether inference of K configurations from arbitrary K rules is possible, and whether it would be consistent with the K configuration abstraction algorithm.

3.9 Sequencing of Side Effects

Following the C++ standard [7], CinK decouples side effects of some constructs to allow delaying memory writes to after an expression value has been returned. This gives compilers more freedom for performing optimisations and during code generation. The newest C++ standard uses a relation *sequenced before* to define how side effects are to be ordered with respect to each other and to value evaluation. The CinK specification uses auxiliary constructs for side effects and uses a bag to collect side effects. An auxiliary sequence point construct forces finalisation of side effects in the bag. We are currently experimenting with funcons to express decoupled side effects.

4 Funcons in K

We now illustrate our K specification of the syntax and semantics of the funcons and value types used in our component-based analysis of CinF. We specify each funcon and value type in a separate module, to facilitate selective reuse. Since modularity is a significant feature of our specifications, we show some of the specified imports. The complete specifications are available online, together with the K specification of the translation of CinF programs to funcons.

4.1 Expressions

Expressions compute values:

```
module EXPR  imports VALUES
  syntax Expr ::= Values
  syntax KResult ::= Values
```

Our specifications of value types lift the usual value operations to expression funcons, each of which is strict in all its arguments:

```
module INTEGERS  imports EXPR ...
  syntax Expr ::= "int-times" "(" Expr "," Expr ")" [strict]
                | ...
  syntax Values ::= Int
  rule int-times(I1:Int, I2:Int) ⇒ I1 *Int I2
  rule ...
```

In contrast, the conditional expression funcon `if-true(E1,E2,E3)` is strict only in *E1*, and its rules involve unevaluated expression arguments:

```
module IF-TRUE-EXPR  imports EXPR ...
  syntax Expr ::= "if-true" "(" Expr "," Expr "," Expr ")" [strict(1)]
  rule if-true(true, E:Expr, _) ⇒ E
  rule if-true(false, _, E:Expr) ⇒ E
```

We specify a corresponding funcon for conditional commands separately, since it appears that K modules cannot have parametric sorts (although the rules above could be generalised to arbitrary *K* arguments).

4.2 Declarations

```
module DECL  imports BINDINGS
  syntax Decl ::= Bindings
  syntax KResult ::= Bindings
```

Bindings are values corresponding to environments (mapping identifiers to values), and come equipped with some operations that can be used to compose declarations:

```
module BINDINGS  imports DECL
  syntax Bindings ::= bindings(Map)
  syntax Decl ::= "bindings-union" "(" Decl "," Decl ")" [strict]
  rule bindings-union(bindings(M1:Map), bindings(M2:Map)) ⇒
    bindings(M1 M2)
```

We could have included the funcon `bind-value(I,E)` as an operation in the above module, since it is strict in its only expression argument:

```
module BIND-VALUE  imports ...
  syntax Decl ::= "bind-value" "(" Id "," Expr ")" [strict(2)]
  rule bind-value(I:Id, V:Values) ⇒ bindings(I |-> V)
```

In contrast, the following funcons involve inspecting or (temporarily) changing the current environment, which is assumed to be in an accompanying cell:

```
module BOUND-VALUE  imports ...
  syntax Expr ::= "bound-value" "(" Id ")"
  rule <k> bound-value(I:Id) ⇒ V:Values ...</k>
       <env>... I |-> V ...</env>

module SCOPE-COMM  imports ...
  syntax Comm ::= "scope" "(" Decl "," Comm ")" [strict(1)]
  rule <k> scope(bindings(Env:Map), C:Comm) ⇒
           reset-env(Env', C) ...</k>
       <env> Env':Map ⇒ Env'[Env] </env>

module ACCUM  imports ...
  syntax Decl ::= "accum" "(" Decl "," Decl ")" [strict(1)]
  rule <k> accum(bindings(Env:Map), D:Decl) ⇒
           reset-env(Env', bindings-union(bindings(Env), D)) ...</k>
       <env> Env':Map ⇒ Env'[Env] </env>
```

The auxiliary operation `reset-env(M,K)` preserves the result of `K` when resetting the current environment to `M`:

```
module RESET-ENV
  syntax K ::= "reset-env" "(" Map "," K ")" [strict(2)]
  rule <k> reset-env(Env:Map, V':KResult) ⇒ V' ...</k>
       <env> _:Map ⇒ Env </env>
```

The `K` argument could be of sort `Expr`, `Decl` or `Comm`. Since we do not use `reset-env` directly in the translation of CinF to funcons, the fact that `reset-env(M,K)` is (semantically) of the same sort as `K` is irrelevant.

4.3 Commands

```
module COMM  imports SKIP
  syntax Comm ::= Skip
  syntax KResult ::= Skip
```

In contrast to the usual style in K specifications, commands compute the unique value `skip`:`Skip` on normal termination, rather than dissolving. However, this difference does not affect the translation of programs to funcons.

```
module SEQ-DECL  imports ...
  syntax Decl ::= "seq" "(" Comm "," Decl ")" [strict(1)]
  rule seq(skip, D:Decl) ⇒ D
```

As with `if-true`, the funcon `seq(C,X)` is essentially generic in `X`, but its syntax needs to be specified separately for each sort of `X`. In contrast, the sort of `effect(X)` is independent of the sort of `X`, and we can specify it generically:

```
module EFFECT  imports COMM
  syntax Comm ::= "effect" "(" K ")" [strict]
  rule effect(_:KResult) ⇒ skip
```

The specification of `while-true` illustrates reuse *between* funcon specifications:

```
module WHILE-TRUE
  imports COMM
  imports EXPR
  imports IF-TRUE-COMM
  imports SEQ-COMM
  syntax Comm ::= "while-true" "(" Expr "," Comm ")"
  rule while-true(E:Expr, C:Comm) ⇒
          if-true(E, seq(C, while-true(E, C)), skip)
```

4.4 Variables

Variables are themselves treated as values:

```
module VARIABLES  imports ...
  syntax Variables ::= "no-variable"
  syntax Values ::= Variables
```

The specifications of the funcons for allocating, assigning to, and inspecting the values stored in variables are much as usual, and we omit them here.

4.5 Functions

```
module FUNCTIONS  imports ...
  syntax Functions ::= "abstraction" "(" Expr ")"
  syntax Values ::= Functions
```

The operation `abstraction(E)` constructs a value from an *unevaluated* expression `E`. It can then be closed to obtain static bindings for identifiers in `E` (the K specification of the funcon `close(E)` is unsurprising, and omitted here).

```
module APPLY  imports ...
  syntax Expr ::= "apply" "(" Expr "," Expr ")" [strict]
  rule apply(abstraction(E:Expr), V:Values) ⇒ supply(V, E)
```

The funcon `supply(E1,E2)` makes the value of `E1` available as '`given`' in the evaluation of `E2`:

```
module SUPPLY-EXPR  imports ...
  syntax Expr ::= "supply" "(" Expr "," Expr ")" [strict(1)]
  rule <k> supply(V:Values, E:Expr) ⇒ reset-given(V', E) ...</k>
       <given> V' ⇒ V </given>

module GIVEN  imports ...
  syntax Expr ::= "given"
  rule <k> given ⇒ V:Values ...</k> <given> V </given>
```

The specifications of the funcons `throw` and `catch` assume that all cells used to represent the current context of a computation are grouped under a unique context cell. This gives improved modularity: the specification remains the same when further contextual cells are required. In other respects, the specification follows the usual style in the K literature, using a stack of exception handlers:

```
module THROW  imports ...
  syntax Comm ::= "throw" "(" Expr ")" [strict]
  rule <k> (throw(V':Values) ~> _) ⇒ (apply(F, V') ~> K) </k>
       <xstack> (F:Functions, K:K, B:Bag) ⇒ . ...</xstack>
       <context> _ ⇒ B </context>

module CATCH  imports ...
  syntax Expr ::= "catch" "(" Comm "," Expr ")" [strict(2)]
  rule <k> (catch(C:Comm, F:Functions) ⇒ (C ~> popx)) ~> K </k>
       <xstack> . ⇒ (F, K, B) ...</xstack>
       <context> B:Bag </context>
  syntax K ::= "popx"
  rule <k> popx ⇒ . </k> <xstack> _:ListItem ⇒ . ...</xstack>
```

Funcons `throw` and `catch` have the most complicated definitions of all, yet they are still modest in size and complexity.

5 Related Work

The work in this paper was inspired by a basic specification of the IMP example language in funcons using K by Roșu. IMP contains arithmetic and boolean expressions, variables, if- and while-statements, and blocks. The translation to funcons is specified directly using K rewrite rules without defining sorted translation functions. The example can be found in the stable K distribution.[4]

CinK, the sublanguage of C++ that we use as a case study in this paper, is taken from a technical report by Lucanu and Șerbănuță [8]. We have limited ourselves to the same subset of C++, except that we omit arrays.

SIMPLE [12] is another K example language which is fairly similar to CinK. The language is presented in two variants: an untyped and a typed one. The definition of typed SIMPLE uses a different syntax and only specifies static semantics. With the component-based approach, we specify a single translation of language constructs to funcons. The MSOS of the funcons defines separate relations for typing and evaluation; in K, it seems we would need to provide a separate static semantics module for each funcon, since the strictness annotations and the computation rules are different.

K specifications scale up to real-world languages, as illustrated by Ellison's semantics of C [4]. The PLanCompS project is currently carrying out major case studies (C#, Java) to examine how the funcon-based approach scales up to large languages, and to test the reusability of the funcon specifications.

Specification of individual language constructs in separate K modules was proposed by Hills and Roșu [6] and further developed by Hills [5, Chapter 5]. They obtained reusable rules by inferring the transformations needed for the rules to match the overall K configuration. The reusability of their modules was limited by their dependence on language syntax, and by the fact that the semantics of individual language constructs is generally more complicated than that of individual funcons.

[4] `http://www.kframework.org`

6 Conclusion

We have given a component-based specification of CinK, using K to define the translation of CinK to funcons as well as the (dynamic) semantics of the funcons themselves. This experiment confirms the feasibility of integrating component-based semantics with the K Framework.

The K specification of each funcon is an independent module. Funcons are significantly simpler than constructs of languages such as CinK, and it was pleasantly straightforward to specify their K rules. However, we would have preferred the K configurations for combination of funcons to be generated automatically.

Many of the funcons used here for CinK were introduced in the component-based specification of Caml Light [3], demonstrating their reusability. The names of the funcons are suggestive of their intended interpretation, so the translation specification alone should convey a first impression of the CinK semantics. Readers are invited to browse the complete K specifications of our funcons online, then compare our translation of CinK to funcons with its direct specification in K [8].

In continuation of this work, we are investigating funcons to specify deferred side-effects and sequence points. We are also aiming to define the static semantics of funcons in K, so our translation would induce a static semantics for CinK.

References

1. F. Chalub and C. Braga. Maude MSOS tool. In *WRLA 2006*, volume 176 of *ENTCS*, pages 133–146. Elsevier, 2007.
2. M. Churchill and P. D. Mosses. Modular bisimulation theory for computations and values. In *FoSSaCS 2013*, volume 7794 of *LNCS*, pages 97–112. Springer, 2013.
3. M. Churchill, P. D. Mosses, and P. Torrini. Reusable components of semantic specifications. In *MODULARITY'14*. ACM, 2014. To appear.
4. C. Ellison and G. Roșu. An executable formal semantics of C with applications. In *POPL'12*, pages 533–544. ACM, 2012.
5. M. Hills. *A Modular Rewriting Approach to Language Design, Evolution and Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 2009.
6. M. Hills and G. Roșu. Towards a module system for K. In *WADT'08*, volume 5486 of *LNCS*, pages 187–205. Springer, 2009.
7. ISO International Standard ISO/IEC 14882:2011(E) – Programming Language C++, 2011. `http://isocpp.org/std/the-standard`.
8. D. Lucanu and T. F. Șerbănuță. CinK – an exercise on how to think in K. Tech. Rep. TR 12-03 (v2), Faculty of Comput. Sci., A. I. Cuza Univ., Dec. 2013.
9. D. Lucanu, T. F. Șerbănuță, and G. Roșu. K Framework distilled. In *WRLA'12*, volume 7571 of *LNCS*, pages 31–53. Springer, 2012.
10. P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.
11. G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
12. G. Roșu and T. F. Șerbănuță. K overview and SIMPLE case study. In *K'11*, ENTCS. Elsevier, 2014. To appear.
13. T. F. Șerbănuță, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Roșu. The K primer (v3.3). In *K'11*, ENTCS. Elsevier, 2014. To appear.

An integration of CafeOBJ into Full Maude*

Adrián Riesco

Facultad de Informática, Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es

Abstract. We present in this paper an integration of CafeOBJ into Full Maude. We have developed a grammar to parse any CafeOBJ specification, an intermediate language to store it, and a translation from this representation into Maude specifications. This integration enhances CafeOBJ functionality in many ways: our intermediate representation has been developed mirroring Maude metalevel, and hence it allows CafeOBJ users to analyze, modify and execute them; CafeOBJ specifications can use Maude commands, including the LTL model checker; other Full Maude tools can be easily combined with this extension; and we provide an alternative implementation for CafeOBJ that can be easily modified and extended. We present here the ideas for parsing and translating CafeOBJ specifications, and illustrate with examples the features listed above.

Keywords: CafeOBJ, Full Maude, Integration, Metalevel

1 Introduction

CafeOBJ [9] is a language for writing formal specifications of models for wide varieties of software and systems, and verifying properties of them. CafeOBJ implements equational logic by rewriting and can be used as a powerful interactive theorem proving system. Specifiers can write proof scores [10] also in CafeOBJ and perform proofs by executing these proof scores. CafeOBJ provides several features to ease the specification of systems. These features include a flexible mix-fix syntax, powerful and clear typing system with ordered sorts, parameterized modules and views for instantiating the parameters, module expressions, operators for defining terms, and equations for defining the (possibly conditional) equalities between terms and (possibly conditional) transitions for specifying how a system evolves, among others. However, only a subset of the CafeOBJ specifications, the equational part, is executable, where the operational semantics is given by a conditional order-sorted term rewriting system.

Maude modules are executable rewriting logic specifications. Maude functional modules [1, Chapter 4] are executable membership equational specifications that allow the definition of sorts; subsort relations between sorts; operators

* Research partially supported by Japanese project Kakenhi 23220002, MICINN Spanish project *StrongSoft* (TIN2012-39391-C04-04), and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC1465).

for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative or commutative, for example; memberships asserting that a term has a sort; and equations identifying terms. Both memberships and equations can be conditional. Maude system modules [1, Chapter 6] are executable rewrite theories. A system module can contain all the declarations of a functional module and, in addition, declarations for rules and conditional rules. An important feature of rewriting logic is that it is reflective, that is, it can be faithfully interpreted in terms of itself. This feature is efficiently implemented in Maude by means of the `META-LEVEL` module [1, Chapter 14], which allows us to use Maude modules and terms as usual data.

Full Maude [1, Part II] is an extension of Maude written in Maude itself. Full Maude provides an even more powerful module algebra than the one available in Core Maude, features for parsing and printing Maude modules, and an explicit module database. This database, combined with the meta-level features explained above, allows us to introduce, remove, modify, and analyze the modules introduced by the user. Moreover, it is also possible to change the syntax of existing features and add new kinds of modules and commands. Full Maude is built on top of the Loop Mode [1, Chapter 17], which provides a mechanism to read the modules and commands introduced by the user enclosed in parentheses, and to show him the results generated by these commands. For these reasons, Full Maude has been traditionally used as a basis for further extensions, either for extra syntactic constructs, like the support for Real-Time modules [14], or for new commands, like the `narrowing` search currently available for symbolic execution [2, Chapter 16].

We present in this paper an extension of Full Maude to parse CafeOBJ modules. The advantages obtained by using this tool, publicly available at `http://maude.sip.ucm.es/cafe/`, are:

- Maude modules can be imported by CafeOBJ modules, and vice versa. The former is specially useful because Maude provides the predefined modules `SATISFACTION`, `LTL-SIMPLIFIER`, and `MODEL-CHECKER` [1, Chapter 12], which allow the user to define and prove LTL properties on CafeOBJ specifications. We can also use the Loop Mode [1, Chapter 17] to develop interactive tools. Moreover, we have defined an intermediate representation of CafeOBJ specifications that mirrors Maude metalevel, and have included functions to execute terms using these modules. That is, CafeOBJ modules can use CafeOBJ modules and terms as standard data, just as several Maude applications have been designed during the last years.
- Maude commands can be used on CafeOBJ specifications. This allows the user to use, among others, the `rew` command to apply transitions (and normalization via equations) to CafeOBJ terms (which cannot be done in the current release of CafeOBJ) [1, Chapter 6]; the `search` command to perform searches to check invariants [1, Chapter 12]; or the `narrowing` command for symbolic execution [2, Chapter 16].
- It provides a new implementation of CafeOBJ. Our interface parses any CafeOBJ module and accepts `open-close` environments, required to exe-

cute proof scores. We also process behavioral specifications, although the current version of the tool does not distinguish between behavioral and non-behavioral statements in the translation.
Moreover, this new implementation is more powerful in the sense that any CafeOBJ programmer can add new syntax and commands. Although this extension would require modifying the Maude code used by the interface, it is so similar to CafeOBJ code that it can be easily understood. Actually, the code has been designed with this feature in mind, so the syntax and parsing modules are carefully distinguished and documented.
As an example of the syntax that can be added to CafeOBJ specifications, our parser allows the user to use matching and rewrite conditions, as well as using the `nonexec` and `metadata` attributes in equations and transitions. Some of these features are available in the latest release of CafeOBJ, while others are only supported by our implementation.

- It allows an easy integration of CafeOBJ specifications with any tool implemented on top of Full Maude. We have currently integrated the Maude Declarative Debugger and Test-case Generator [16] and the Constructor-based Inductive Theorem Prover [11]. Our goal when integrating these tools was to provide a minimum framework where CafeOBJ functions can be tested, fixed when a wrong behavior is found, and proved correct with respect to some properties, once we have confidence in the soundness of the implementation. However, many other interesting tools can be integrated using our approach.
- Finally, we provide a script to connect CafeOBJ with Full Maude in a transparent way. We have implemented a Java class that transforms the source code to meet the format required by Full Maude, which includes enclosing the modules in parentheses, adding the ‘ to escape characters such as `[`, `]`, or `,`, and removing CafeOBJ comments, among others. In this way, it is not necessary to modify the original CafeOBJ specifications to use the interface.

The rest of the paper is organized as follows: Section 2 briefly introduces the related work, while Section 3 present the basic notions used throughout the paper. Section 4 describes the parsing and translation process. Section 5 illustrates how to use the tool. Finally, Section 6 presents the concluding remarks and outlines some lines of future work.

2 Related Work

The most similar examples to the present work are Full Maude itself [1, Part II], Real-Time Maude [14], and the Maude Strategy Language [7]. The former defines a complete syntax for Maude, extends it with support for object-oriented modules, and provides commands to execute them. Similarly, Real-Time Maude defines real-time modules and timed commands to execute them, while the Strategy Language extends Maude modules with syntax for defining execution strategies, as well as rewrite commands using these strategies. Our work follows the same steps: it requires to define the syntax of our modules and commands,

parse them, translate them into Maude (in Full Maude this is only the case for object-oriented modules, since standard modules do not require translation), and execute the commands. Nonetheless, we take advantage of many features developed for Full Maude and reused later [5] that greatly ease the parsing task.

Besides these tools, Maude has been used as a semantical framework to specify the semantics of several languages, such as LOTOS [17], CCS [17], or C [8]. These researches, as well as several other efforts to describe a methodology to represent the semantics of programming languages in Maude, led to the *rewriting logic semantics project* [12], which presents a comprehensive compilation of these works.

Another translation from CafeOBJ to Maude can be found in [18]. There, the authors translate a subset of CafeOBJ specifications (more specifically, specifications of state machines standing for asynchronous distributed systems) into Maude to perform model checking. Although they follow an approach similar to the one in the current paper, it is focused in just one kind of specification, and hence it lacks scalability.

3 Preliminaries

We present in this section some basic notions required throughout the rest of the paper. First, we describe CafeOBJ and Maude by means of an example. Then, we give some details about the Maude metalevel and Full Maude.

3.1 CafeOBJ and Maude

CafeOBJ (on the lefthand side) can define modules with loose semantics by using the syntax `mod*`. For example, we can define a module `ELT` requiring the existence of a sort `Elt` and an element of this sort, called `mt`, which is a constructor. This kind of behavior is specified in Maude (on the righthand side) as a theory:

```
mod* ELT {
 [Elt]
 op mt : -> Elt {constr}
}
```

```
fth ELT is
 sort Elt .
 op mt : -> Elt [ctor] .
endfth
```

We can use this module to define a parameterized module with tight semantics, with syntax `mod!`. The module `LIST` below indicates that it receives a parameter `X` fulfilling the requirements stated by `ELT`. This module first defines the sort `List` for lists. Similarly, we define a parameterized system module `LIST` in Maude with syntax `mod`:

```
mod! LIST(X :: ELT) {
  [List]
```

```
mod LIST{X :: ELT} is
 sort List .
```

The constructors are defined, as shown above, with the keyword `op` and the `constr` attribute. In this case the constructors are `nil` for empty lists and the juxtaposition for placing an element of sort `Elt` in front of a list. Note the different syntax for the sort `Elt`, qualified by the parameter `X`:

```
op nil : -> List {constr}                op nil : -> List [ctor] .
op __ : Elt.X List -> List {constr}      op __ : X$Elt List -> List [ctor] .
```

We can also define functions for lists. For example, composition of lists is defined by distinguishing constructors on the first argument. Note that both CafeOBJ and Maude follow the same syntax, although CafeOBJ allows some extra syntactic sugar, including just-once on-the-fly declaration of variables:

```
var E : Elt.X      var L : List          var E : X$Elt .  var L : List .
op _@_ : List List -> List               op _@_ : List List -> List .
eq [c1] : nil @ L = L .                  eq [c1] : nil @ L = L .
eq [c2] : (E L) @ L':List =              eq [c2] : (E L) @ L':List =
          E (L @ L') .                             E (L @ L':List) .
```

Similarly, we can define the `reverse` function. This function uses the constant `mt` from module `ELT` as the reverse of the empty list,[1] while the reverse for bigger lists is defined as usual by using the composition above:

```
  op reverse : List -> List              op reverse : List -> List .
  eq [r1] : reverse(nil) = mt nil .      eq [r1] : reverse(nil) = mt nil .
  eq [r2] : reverse(E L) =               eq [r2] : reverse(E L) =
            reverse(L) @ (E nil) .                 reverse(L) @ (E nil) .
```

We can also define non-deterministic transitions. For example, we can combine two lists by using the commutative operator `mix` and two transitions to indicate that the next element is the first one of any of the lists (thanks to the matching module commutativity):

```
  op mix : List List -> List {comm}    op mix : List List -> List [comm] .
  trans [m1] : mix(nil, L) => L .      rl [m1] : mix(nil, L) => L .
  trans [m2] : mix(E L, L')            rl [m2] : mix(E L, L')
            => E mix(L, L') .                    => E mix(L, L') .
}                                    endm
```

Finally, in CafeOBJ we can use an on-the-fly view to instantiate `LIST` with natural numbers:

```
mod! NAT-LIST {
  pr(LIST(view to NAT {sort Elt -> Nat, op mt -> 0}))
}
```

On the other hand, we need to define an explicit view in Maude, and then use this view to instantiate the module:

```
view Nat from Elt to NAT is
 sort Elt to Nat .
 op mt to 0 .
endv

mod NAT-LIST is
 pr LIST{Nat} .
endm
```

[1] This is a wrong definition that will be detected and fixed in Section 5.3.

3.2 Maude Metalevel and Full Maude

Exploiting the fact that rewriting logic is reflective [3], an important feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [1, Chapter 14], a characteristic that allows many advanced metaprogramming and metalanguage applications. In this work, we take advantage of this feature to parse, store, transform, and execute CafeOBJ modules.

Full Maude [1, Part II] is an extension of Maude written in Maude itself. Full Maude is built on top of the `LOOP-MODE` module [1, Chapter 17]. This module allows input/output interaction by means of the `[_,_,_]` operator, which builds terms of sort `System` and where the first argument corresponds to the input introduced by the user, which must be enclosed in parentheses to be recognized; the second one is a term of sort `State` that can be defined by the user for each application; and the third one the output shown to the user.

In Full Maude this `State` is defined by using a class `Database`, which has an attribute `db` standing for the Full Maude database. It also has attributes for the current `input`, the `output` not processed yet, and the `default` module. Essentially, the Loop Mode transforms the data introduced by the user into a list of quoted identifiers; this list is then meta-parsed by Full Maude by using the `GRAMMAR` module, which includes the syntax for modules and commands. If this parsing is successful, then the term thus obtained is placed in the `input` attribute. Different inputs are treated by using rules: modules and views are processed to check whether they fulfill the semantic constraints required by Maude, and then introduced into the database, while commands are executed by using this database. The results must be placed in the `output` attribute; a rule will move this data to the third component of the system.

Hence, our aims in this paper are to extend `GRAMMAR` to include CafeOBJ syntax, process the new terms obtained from the parsing, and define commands (and the appropriate rules) to deal with these new features.

4 Introducing CafeOBJ Modules into the Full Maude Database

We present in this section the basic ideas to introduce CafeOBJ modules into the Full Maude database. First, we describe how CafeOBJ modules are parsed. Then we show how the obtained modules can be translated into Maude and used by other tools implemented in Full Maude.

4.1 Parsing CafeOBJ Modules and Commands

As explained in the previous section, in order to parse CafeOBJ modules we have to define its syntax, which will be used by Full Maude to create a term that will be processed to obtain the actual module. We use the metarepresentation of this module to extend the `GRAMMAR` metamodule from Full Maude, providing the metamodule `CafeGRAMMAR`. It can be used to parse both Maude and CafeOBJ modules and commands.

Basically, the syntax follows the CafeOBJ grammar in [13], although we have extended it with some features that will be available in the next release of CafeOBJ, such as the `nonexec` attribute or matching conditions. Following the standard approach, we define a sort for each syntactic category in the grammar, and operator declarations for each production rule. In this way, we specify a module `CafeMETA-SIGN` where this information is contained. For example, the sort `@CafeTransDecl@`[2] stands for the definition of transitions in CafeOBJ syntax:

```
op trans_=>_. : @CafeBubble@ @CafeBubble@ -> @CafeEqDecl@ [ctor] .
op ctrans_=>_if_. : @CafeBubble@ @CafeBubble@ @CafeBubble@
                  -> @CafeTransDecl@ [ctor] .
op ctrans_=>_if_. : @CafeBubble@ @CafeBubble@ @CafeBubble@
                  -> @CafeTransDecl@ [ctor] .
```

Note that we use a special sort `@CafeBubble@` to encapsulate terms that can take any form. Basically, a bubble is any list of quoted identifiers, which must be later parsed to obtain a valid term in the current module.

These declarations, as well as the rest of declarations for the statements available in a CafeOBJ module, are defined as a subsort of a `@CafeDeclList@`, which are composed by means of a juxtaposition operator:

```
subsorts @CafeImportDecl@ ... @CafeTransDecl@ < @CafeDeclList@ .
op __ : @CafeDeclList@ @CafeDeclList@ -> @CafeDeclList@ [assoc] .
```

For example, the transition `m1` from Section 3 would be parsed as:

```
'trans_=>_.['CafeBubble['__[''[.Qid, ''m1.Qid, ''], ':, ''mix.Qid,
    ''(.Qid, ''nil.Qid, ''(,, ''L.Qid, '').Qid]], 'CafeBubble[''L.Qid]]
```

Note that the label is included in the bubble for the lefthand side; it must be extracted before processing this side (analogously, attributes might appear in the bubble for the righthand side). This term must be now parsed again in order to check whether it fulfill the semantics constraints, e.g., the terms only use variables previously defined, they are bound either in the lefthand side or in a matching condition, and terms are built using existing operators. This second phase returns, when the module is correct, a term of sort `CafeModule`:

```
op mod*_{__[_]____} : CafeHeader CafeImportList HiddenSortDecl SortSet
 CafeSubsortDeclSet CafeOpDeclSet CafeEqSet CafeTransSet
 -> CafeModule [ctor] .
```

Our definition of CafeOBJ modules uses the sorts `Qid`, `Term`, and `Condition` from Maude metalevel to define the sorts used here. For example, transitions are declared as follows:

```
op trans_=>_{_}. : Term Term CafeAttrSet -> CafeTrans [ctor] .
op ctrans_=>_if_{_}. : Term Term Condition CafeAttrSet -> CafeTrans [ctor] .
```

[2] We follow the Full Maude convention and enclose sorts for parsing in `@`.

In this way, the transition `m1` is represented as:

```
trans 'mix['nil.List, 'L:List] => 'L:List {label:('m1)} .
```

Once the final module has been obtained, it is stored in a database, which is just a partial function from quoted identifiers (of sort `Qid`) to `CafeModule`. This modules can be retrieved, modified, executed, and stored again, as we will see in Section 5. Note that the current version of the tool does not support metasyntax for views; they are just introduced as Maude views.

Regarding commands, we provide the syntax for `open...close` environments, which combine operator declarations (mainly constants) and equation definitions with `red` commands to define proof scores [10], and specific commands for dealing with CafeOBJ modules. In this case we create a module on-the-fly, where the reductions take place.

4.2 Translating the Modules

Taking advantage of the similarities between the syntax and the semantics of CafeOBJ and Maude, most of the transformations performed by our tool are straightforward. Both languages have modules with loose semantics (called *theories* in Maude), modules with tight semantics, parameterized modules, views to instantiate these modules, equations, and transitions (*rules* in Maude) as main features. From the Maude point of view there are some features that cannot be translated into CafeOBJ, being the main one the membership axioms stating the members of a sort, because Maude implements membership equational logic while the CafeOBJ type system is based on order sorted algebra. However, the differences in this case are not important because we are interested in the translation from CafeOBJ to Maude.

There are two important features in CafeOBJ that cannot be translated into Maude. Both of them are related to the modules with loose semantics: (i) these modules can be parameterized in CafeOBJ but cannot be parameterized in Maude and (ii) these modules can be imported in any mode (being the modes `protecting`, indicating that no junk and no confusion is added to the sorts; `extending`, denoting that no confusion is allowed; and `including`, indicating that there are no restrictions, see [1, Chapter 8] for details), while Maude theories can only be imported in `including` mode. We have dealt with these restrictions in a conservative way. First, we translate these modules, that should be Maude theories, as modules (i.e., they have tight semantics), and a warning message is shown. This change is harmless if our aim is to execute them or to use any of the tools currently integrated (the declarative debugger and the CITP), but has two disadvantages: (a) it might fail later, if this module is used as the target of a view, and (b) other tools, not integrated yet, might distinguish between the different kinds of modules. Similarly, we always translate the importation modes for these modules as `including`, which is also fine in our case (the tools integrated thus far use flattened modules) but might produce problems with other tools. The user can force the tool to translate the modules without modifications with the command `strict translation on .`).

There are also some other complex features that require a non-straightforward translation. More specifically, the CafeOBJ syntax for views is much more flexible than the one used by Maude: they can be defined on-the-fly and can be used in an order different from the one specified in the parameterized module by using the parameter name. The former is solved by creating explicit views with fresh view identifiers, while the latter requires to manipulate the parameterized module from the database to reorder the views.

Basically, our implementation defines a function `cafe2maude`, which takes a `CafeModule` and returns a Maude `Module`:

```
op cafe2maude : CafeModule -> Module .
```

It uses auxiliary functions to translate each element in a CafeOBJ module. For instance, transitions are translated into rules as follows:

```
op cafe2maude : CafeTrans -> Rule .
eq cafe2maude(trans T => T' {AtS} .) = rl T => T' [cafe2maude*(AtS)] . .
eq cafe2maude(ctrans T => T' if C {AtS} .) = crl T => T'
                                              if C [cafe2maude*(AtS)] . .
```

where `cafe2maude*` is an auxiliary function that translates the attributes.

As explained in Section 3.2, the connection between the Loop Mode and the behavior of the tool is implemented by rules. We have defined a new class `CafeDatabase`, subclass of `Database`, to take care of the translation and the new commands:

```
sort CafeDatabaseClass .
subsort CafeDatabaseClass < DatabaseClass .
op CafeDatabase : -> CafeDatabaseClass [ctor] .
```

This class defines two new attributes: `strict`, which indicates whether the translation is strict or not, and `cafeDB`, which contains the CafeOBJ database:

```
op strict :_ : Bool -> Attribute [ctor] .
op cafeDB :_ : CafeDB -> Attribute [ctor] .
```

4.3 Combining CafeOBJ and other Full Maude tools

Using the modules described in the previous sections, it is easy to modify any tool built in Full Maude for Maude specifications and make it work with CafeOBJ modules, given that they follow two standard principles:[3]

– They use a module extending `GRAMMAR` to parse their modules/commands. In this case, it is enough to extend `CafeGRAMMAR` instead, and CafeOBJ modules will be parsed.

[3] Note that these changes will allow us to execute the tools. However, some theoretical considerations may be required to prove that this execution is correct.

– They define a subclass of `Database` to process their modules/commands. We have to modify this definition to extend `CafeDatabase`. It is also required to initialize the attributes `strict` and `cafeDB`, so they can be used later.

To test the benefits of this approach we have already worked with the Maude declarative debugger and test-case generator [16] and the Constructor-based Inductive Theorem Prover (CITP) [11]. The main problem of the integration is that the output provided by the tool refers to the transformed Maude code. Although this might be fine in some cases (e.g. the debugger refers to the label of the wrong statement, so it is safe to use it, see Section 5.3 for details), in some others it is interesting to refer to the original CafeOBJ module or just use commands which are specifically defined for CafeOBJ users. In this case, some extra changes are required, as shown in the next section for the CITP.

5 Connecting CafeOBJ and Maude

We present in this section how to use the most important features of our implementation. We first show how to use the metalevel representation of CafeOBJ. Then, we describe the basic commands provided in the interface and how to use the Maude Declarative Debugger and the Constructor-based Inductive Theorem Prover. All the modules, scripts, and examples shown here are available at `http://maude.sip.ucm.es/cafe/`.

5.1 Metaprogramming in CafeOBJ

We provide in the `META-CAFE-SYNTAX` module the syntax for CafeOBJ modules. It follows the syntax in the predefined module `META-LEVEL` for Maude modules, but uses specific syntax to follow CafeOBJ conventions. These modules are retrieved and inserted from/into the database with the functions `getTopModule` and `setTopModule`. Note that, since these modules are stored in a specific attribute of the `CafeDatabase` class, specifications using the database are not completely transparent from Maude syntax:

```
op getTopModule : CafeDB Qid ~> CafeModule .
op setTopModule : CafeDB Qid CafeModule -> CafeDB .
```

Finally, these modules can be modified and executed by using the functions in `CAFE-META-LEVEL`. It includes functions for accessing the different components of a module, update them, and for executing terms in a given modules. The current version of the tool provides the functions `metaReduce`, for applying equations until a normal form is reached; `metaRewrite`, for applying transitions given a bound in the number of transitions applied; and `metaFrewrite`, for fair application of transitions given a bound in the number of transitions applied and the maximum number of rewrites at each entitled position on each traversal of a subject term (see [1, Chapter 14] for details):

```
op metaReduce : Qid Term CafeDB Database -> ResultPair .
op metaRewrite : Qid Term Bound CafeDB Database -> ResultPair .
op metaFrewrite : Qid Term Bound Nat CafeDB Database -> ResultPair .
```

Note that these functions require the Maude database, since they might import some Maude modules. They are implemented by building the corresponding flat Maude module and then using the appropriate built-in Maude functions.

For example, we could define a function `getCommOps` extracting the commutative operators from a CafeOBJ module by using an auxiliary function `filterCommOps` that keeps the commutative operators from a set:

```
op getCommOps : CafeModule -> CafeOpDeclSet
eq getCommOps(CM) = filterCommOps(getOps(CM)) .
op filterCommOps : CafeOpDeclSet -> CafeOpDeclSet
eq filterCommOps(none) = none .
eq filterCommOps(COD CODS) = if isComm?(COD) then COD
                             else none fi filterCommOps(CODS) .
```

where `isComm?` is an auxiliary function that checks whether an operator is commutative. Note that we allow operators with both the `op` definition and the `pred` keyword. This function uses another auxiliary function `containsComm?` which just traverses the attributes looking for `comm`:

```
pred isComm? : CafeOpDecl
eq isComm?(op Q : TyL -> Ty {AtS}) = containsComm?(AtS) .
eq isComm?(pred Q : TyL {AtS}) = containsComm?(AtS) .
pred containsComm? : CafeAttrSet
eq containsComm?(none) = false .
eq containsComm?(A AtS) = A == comm or containsComm?(AtS) .
```

5.2 Basic Commands

Once the files in the webpage have been downloaded and the paths have been configured, and assuming the modules above are saved in a file called `wrla.cafe`, we can start the tool by typing:

```
$ ./cafe2maude wrla.cafe
```

The `cafe2maude` script creates a temporary file generated by a Java application. This file contains the original CafeOBJ modules modified in order to be accepted by Full Maude (e.g. adding the parentheses enclosing modules and views, removing CafeOBJ comments, and adding the ‘ character to the escape characters such as `{` or `}`). Once the script is executed, the modules are introduced into the Full Maude database and we can use any Maude command on them. For example, the `rew` command uses transitions to evaluate terms. Note that this command, as well as the one below, is not available in CafeOBJ:

```
Maude> (rew mix(1 3 nil, 2 4 nil) .)
result List : 1 2 3 4 nil
```

We can also use symbolic search to start with terms with variables and look for substitutions that fulfill the conditions imposed by the search. For example, we can look for the term required in the `mix` operator to obtain the result from the `rew` command:

```
Maude> (search [1] mix(L:List, 2 4 nil) ~>! 1 2 3 4 nil .)
Solution 1
L:List --> 1 3 nil
No more solutions.
```

where the `!` option indicates that we are looking for *final* terms and `~>!` distinguishes the symbolic search from the standard one, performed with `=>!`. In this case we obtain the substitution `L:List --> 1 3 nil`, indicating that we needed this list to obtain the result.

Besides using Maude commands, we can also work with CafeOBJ specifications. For example, we can see the original module and execute proof scores. Basically, proof scores are scripts defining an inductive proof, where constants can be declared by means of operators and hypothesis by using equations. The base and the inductive steps are proved by using the `red` command. For example, we can prove the associativity of the `_+_` function as follows:

```
open NAT + BOOL
 ops i j k : -> Nat
 red (0 + j) + k == 0 + (j + k) .       -- base step
 eq (i + j) + k = i + (j + k) .         -- induction hypothesis
 red (s(i) + j) + k == s(i) + (j + k) . -- inductive step
close
```

Once we load the file with this `open-close` environment, Maude executes the `red` commands and provides the following result:

```
Processing open-close environment:
reduce(0 + j)+ k == 0 + j + k .
Result: true : Bool
reduce(s i + j)+ k == s i + j + k .
Result: true : Bool
```

5.3 Using the Declarative Debugger and Test-case Generator

To start this tool it is enough to download the script `cdd`, configure the paths, and execute it with the files we want to test and debug. Then, we can use all the commands described in `http://maude.sip.ucm.es/debugging/` to test and debug our CafeOBJ modules. For example, we can test the `reverse` function by using the so called *function coverage* criterium, which generates ground test cases that must use all the equations defined for `reverse` (`r1` and `r2`) in all the calls (the single call to this function is located in `r2`). This is done by using:

```
Maude> (function coverage .)
Function Coverage selected
Maude> (test in NAT-LIST : reverse .)
1 test cases have to be checked by the user:
    1. The term reverse(0 0 nil) has been reduced to 0 0 0 nil
All calls were covered.
```

That is, the call `reverse(0 0 nil)` uses both `r1` and `r2` for the recursive call (`r2` for the first call and `r1` for the second one). Note that the result of this call is unexpected, because it should also be `0 0 nil`. Hence, this function is buggy and must be debugged. We can do it by typing:

```
Maude> (invoke debugger with user test case 1 .)
Declarative debugging of wrong answers started.
```

This command starts the declarative debugger. Declarative debuggers find bugs in programs by asking questions to the user, that must answer `yes` or `no` (check the webpage above for more possible answers) until the bug is found. Hence, the debugger presents the following question:

```
Is this reduction (associated with the equation r2) correct?
reverse(0 nil) -> 0 0 nil
Maude> (no .)
```

This result is erroneous for the same reasons explained above, so the user answers `no` and the debugging session continues with the following questions:

```
Is this reduction (associated with the equation com2) correct?
(0 nil) @ 0 nil -> 0 0 nil
Maude> (yes .)
Is this reduction (associated with the equation r1) correct?
reverse(nil) -> 0 nil
Maude> (no .)
```

We answer `yes` for a correct composition but `no` for another application of `reverse`. With this information the debugger is able to find the bug:

```
The buggy node is: reverse(nil) -> 0 nil
with the associated equation: r1
```

In fact, the equation `r1` should return just `nil`. The questions asked during the session correspond to the nodes of a tree representing the wrong computation. This tree, which might be useful to the user to check the relations between the calls, can also be shown.

5.4 Using the Constructor-based Inductive Theorem Prover

We have extended the CITP to work with CafeOBJ-like commands, hence obtaining a tool fully customized for CafeOBJ. This has been done by adding an

extra attribute `language` to the tool, which allows us to distinguish between interfaces, while the underlying modules dealing with proofs are left unmodified.

The CITP allows the user to prove properties on CafeOBJ specifications. It is started by the `citp` script. Since we want to prove properties on CafeOBJ specifications, we have to indicate it with a specific command, which sets the `language` attribute explained above to `cafeOBJ`, hence modifying the syntax and the display options to work with CafeOBJ specifications:

```
Maude> (cafeOBJ language .)
CafeOBJ selected as current specification language.
```

Now we can introduce goals, which are depicted as equations or transitions. For example, we can prove the associativity of list composition, using on-the-fly declaration of variables from CafeOBJ, by typing:

```
Maude> (goal NAT-LIST |- eq L1:List @ (L2:List @ L3:List) =
                            (L1 @ L2) @ L3 ;)
============================ GOAL 1-1 ============================
< Module NAT-LIST is concealed ... end,
  eq L1:List @(L2:List @ L3:List) = (L1:List @ L2:List)@ L3:List . >
unproved
INFO: an initial goal generated!
```

This goal can easily be proved by using induction on `L1` and then applying the default tactic with the `auto` command:

```
Maude> (set ind on L1:List .)
INFO: Induction will be conducted on L1:List
Maude> (auto .)
INFO: Goal 1-1 was successfully proved by applying tactic: SI CA CS TC IP
INFO: PROOF COMPLETED
```

It is also possible to state goals involving transitions. For example, we can define the following trivial goal, which just uses the commutativity attribute:

```
Maude> (goal NAT-LIST |- trans mix(L:List, nil) => L ;)
============================ GOAL 1-1 ============================
< Module NAT-LIST is concealed ... end, trans mix(L:List,nil) => L:List . >
unproved
INFO: an initial goal generated!
```

Note that CafeOBJ syntax is used for both the goal and the displayed information. This simple goal can be discarded by just using `auto`:

```
Maude> (auto .)
INFO: Goal 1-1 was successfully proved by applying tactic: SI CA CS TC IP
INFO: PROOF COMPLETED
```

Much more information on the CITP, including several other commands, all of them now customized for CafeOBJ specifications, is described at `http://www.jaist.ac.jp/~danielmg/citp.html`.

6 Concluding Remarks and Ongoing Work

We have presented in this paper a tool to introduce CafeOBJ specifications into the Full Maude database. This tool allows us to use Maude modules and commands with CafeOBJ specifications, provides an implementation of a CafeOBJ metalevel, and eases the task of connecting CafeOBJ specifications with tools implemented on top of Full Maude. Using this feature we provide an environment where CafeOBJ specifications can be tested, debugged, and proved correct by integrating the Maude Declarative Debugger and Test-case Generator and the Constructor-based Inductive Theorem Prover.

We want to improve the implementation of the metalevel in two different ways: first, we want to define the syntax for representing views, in such a way that they can also be analyzed and modified. On the other hand, we are interested in defining more execution commands: currently only `metaReduce`, `metaRewrite`, and `metaFrewrite` are available, but several others can be implemented using our translation for CafeOBJ specifications and the built-in commands in Maude metalevel. Another interesting topic would be distinguish between behavioral and non-behavioral specifications when translating and executing the modules.

We are currently working to extend our framework with the Maude Formal Environment (MFE) [6]. This environment allows to check properties such as termination, confluence, and coherence on Maude specifications. It also includes the Inductive Theorem Prover [4], a tool to prove inductive properties on equational Maude specifications. Integrating this environment with CafeOBJ specifications would allow us to check that the executability requirements hold.

We are also interested in integrating Real-Time Maude [14] in our framework. This integration would be specially interesting for CafeOBJ users, since several protocols, such as [15], has already been specified in CafeOBJ. However, this integration is not straightforward, since it requires to extend the syntax of CafeOBJ specifications with timed rules, as originally implemented for Maude.

Besides connecting more tools, we are also interested in extending the commands for CafeOBJ. More specifically, we are interested in the `t1 =(m,n)=> t2` predicate, which indicates that the term `t2` is reachable from `t1`, with `m` the number of searched terms and `n` the depth of the search (both numbers can be set to `*` to indicate that it is unbounded). This predicate, that is not documented and allows several extra conditions to constrain the states, is similar to the `search` command in Maude. It is interesting to implement this predicate, since it would increment the amount of CafeOBJ commands supported by our interface while providing a documented version in terms of Maude.

References

1. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, *LNCS* 4350. Springer, 2007.

2. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.6)*, January 2011. `http://maude.cs.uiuc.edu/maude2-manual`.
3. M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.
4. M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006. Programming and Languages. Special Issue with Extended Versions of Selected Papers from PROLE 2005: The 5th Spanish Conference on Programming and Languages.
5. F. Durán and P. C. Ölveczky. A guide to extending full maude illustrated with the implementation of Real-Time Maude. In G. Roşu, editor, *Proceedings of the 7th International Workshop on Rewriting Logic and its Applications, WRLA 2008*, *ENTCS* 238(3), pages 83–102. Elsevier, 2009.
6. F. Durán, C. Rocha, and J. M. Álvarez. Towards a Maude formal environment. In G. Agha, J. Meseguer, and O. Danvy, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, *LNCS* 7000, pages 329–351. Springer, 2011.
7. S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In M. Archer, T. B. de la Tour, and C. A. M. noz, editors, *Proceedings of the 6th International Workshop on Strategies in Automated Deduction (STRATEGIES 2006)*, *ENTCS* 174, pages 3–25. Elsevier, 2007.
8. C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages, POPL 2012*, pages 533–544. ACM, 2012.
9. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
10. K. Futatsugi, D. Gâinâ, and K. Ogata. Principles of proof scores in CafeOBJ. *Theoretical Computer Science*, 464:90–112, 2012.
11. D. Gâinâ, M. Zhang, Y. Chiba, and Y. Arimoto. Constructor-based inductive theorem prover. In R. Heckel and S. Milius, editors, *Proceedings of the 5th International Conference in Algebra and Coalgebra in Computer Science, CALCO 2013*, *LNCS* 8089, pages 328–333. Springer, 2013.
12. J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.
13. A. T. Nakagawa, T. Sawada, and K. Futatsugi. *CafeOBJ User's Manual (version 1.4.8)*, July 2010. `http://www.comp.dit.ie/pbrowne/compfund2/manual.pdf`.
14. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20:161–196, 2007.
15. I. Ouranos, K. Ogata, and P. S. Stefaneas. Formal analysis of TESLA protocol in the Timed OTS/CafeOBJ method. In T. Margaria and B. Steffen, editors, *ISoLA (2)*, *LNCS* 7610, pages 126–142. Springer, 2012.
16. A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. Declarative debugging of rewriting logic specifications. *Journal of Logic and Algebraic Programming*, 81(7-8):851–897, 2012.
17. A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67:226–293, 2006.
18. M. Zhang and K. Ogata. Modular implementation of a translator from behavioral specifications to rewrite theory specifications. In B. Choi, editor, *Proceedings of the 9th International Conference on Quality Software, QSIC 2009*, pages 406–411. IEEE Computer Society, 2009.

Rewriting Modulo SMT and Open System Analysis

Camilo Rocha[1], José Meseguer[2], and César Muñoz[3]

Escuela Colombiana de Ingeniería, Bogotá, Colombia
University of Illinois at Urbana-Champaign, Urbana IL, USA
NASA Langley Research Center, Hampton VA, USA

Abstract. This paper proposes *rewriting modulo SMT*, a new technique that combines the power of SMT solving, rewriting modulo theories, and model checking. Rewriting modulo SMT is ideally suited to model and analyze infinite-state *open systems*, i.e., systems that interact with a non-deterministic environment. Such systems exhibit both internal non-determinism, which is proper to the system, and external non-determinism, which is due to the environment. In a reflective formalism, such as rewriting logic, rewriting modulo SMT can be reduced to standard rewriting. Hence, rewriting modulo SMT naturally extends rewriting-based reachability analysis techniques, which are available for closed systems, to open systems. The proposed technique is illustrated with the formal analysis of a real-time system that is beyond the scope of timed-automata methods.

1 Introduction

Symbolic techniques can be used to represent possibly infinite sets of states by means of symbolic constraints. These techniques have been developed and adapted to many other verification methods such as SAT solving, Satisfiability Modulo Theories (SMT), rewriting, and model checking. A key open research issue of current symbolic techniques is extensibility. Techniques that combine different methods have been proposed, e.g., decision procedures [28, 29], unifications algorithms [7, 11], theorem provers with decision procedures [1, 10, 32], and SMT solvers in model checkers [3, 18, 27, 36, 38]. However, there is still a lack of general extensibility techniques for symbolic analysis that simultaneously combine the power of SMT solving, rewriting- and narrowing-based analysis, and model checking.

This paper proposes a new symbolic technique that seamlessly combines rewriting modulo theories, SMT solving, and model checking. For brevity, this technique is called *rewriting modulo SMT*, although it could more precisely be called *rewriting modulo SMT+B*, where B is an equational theory having a matching algorithm. It complements another symbolic technique combining narrowing modulo theories and model checking, namely narrowing-based reachability analysis [8, 26]. Neither of these two techniques subsumes the other.

Rewriting modulo SMT can be applied to increase the power of equational reasoning, e.g., [16, 17, 21], but its full power, including its model checking capabilities, is better exploited when applied to concurrent open systems. Deterministic systems can be naturally specified by equational theories, but specification of concurrent, non-deterministic

systems requires rewrite theories [24], i.e., triples $\mathcal{R} = (\Sigma, E, R)$ with (Σ, E) an equational theory describing system states as elements of the initial algebra $\mathcal{T}_{\Sigma/E}$, and R rewrite rules describing the system's local concurrent transitions. An *open system* is a concurrent system that interacts with an external, non-deterministic environment. When such a system is specified by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, it has two sources of non-determinism, one internal and the other external. Internal non-determinism comes from the fact that in a given system state different instances of rules in R may be enabled, and the local transitions thus enabled may lead to completely different states. What is peculiar about an open system is that it also has external, and often infinitely-branching, non-determinism due to the environment. That is, the state of an open system must include the state changes due to the environment. Technically, this means that, while a system transition in a closed system can be described by a rewrite rule $t \rightarrow t'$ with $vars(t') \subseteq vars(t)$, a transition in an open system is instead modeled by a rule of the form $t(\overrightarrow{x}) \rightarrow t'(\overrightarrow{x}, \overrightarrow{y})$, where $\overrightarrow{y}$ are fresh new variables. Therefore, a substitution for the variables $\overrightarrow{x} \uplus \overrightarrow{y}$ decomposes into two substitutions, one, say θ, for the variables $\overrightarrow{x}$ under the control of the system and another, say ρ, for the variables $\overrightarrow{y}$ under the control of the environment. In rewriting modulo SMT, such open systems are described by conditional rewrite rules of the form $t(\overrightarrow{x}) \rightarrow t'(\overrightarrow{x}, \overrightarrow{y})$ **if** ϕ, where ϕ is a constraint solvable by an SMT solver. This constraint ϕ may still allow the environment to choose an infinite number of substitutions ρ for $\overrightarrow{y}$, but can exclude choices that the environment will never make.

The non-trivial challenges of modeling and analyzing open systems can now be better explained. They include: (1) the enormous and possibly infinitary non-determinism due to the environment, which typically renders finite-state model checking impossible or unfeasible; (2) the impossibility of executing the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ in the standard sense, due to the non-deterministic choice of ρ; and (3) the, in general, undecidable challenge of checking the rule's condition ϕ, since without knowing ρ, the condition $\phi\theta$ is non-ground, so that its E-satisfiability may be undecidable. As further explained in the paper, challenges (1)–(3) are all met successfully by rewriting modulo SMT because: (1) states are represented not as concrete states, i.e., ground terms, but as symbolic constrained terms $\langle t \,;\, \varphi \rangle$ with t a term with variables ranging in the domains handled by the SMT solver and φ an SMT-solvable formula, so that the choice of ρ is avoided; (2) rewriting modulo SMT can symbolically rewrite such pairs $\langle t \,;\, \varphi \rangle$ (describing possibly infinite sets of concrete states) to other pairs $\langle t' \,;\, \varphi' \rangle$; and (3) decidability of $\phi\theta$ (more precisely of $\varphi \wedge \phi\theta$) can be settled by invoking an SMT solver.

Rewriting modulo SMT can be integrated with model-checking by exploiting the fact that rewriting logic is reflective [15]. Hence, rewriting modulo SMT can be reduced to standard rewriting. In particular, all the techniques, algorithms, and tools available for model checking of closed systems specified as rewrite theories, such as Maude's search-based reachability analysis [14], become directly available to perform symbolic reachability analysis on systems that are now infinite-state.

The technique proposed in this paper is illustrated with the formal analysis of the CASH scheduling protocol [13]. This protocol specifies a real-time system whose formal analysis is beyond the scope of timed-automata [2].

2 Preliminaries

Notation on terms, term algebras, and equational theories is used as in [6, 19].

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 $w, w' \in S^*$ and $s, s' \in S$, if $f : w \longrightarrow s$ and $f : w' \longrightarrow s'$ are in F, 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, $[s] = [s]_{\equiv_{\leq}}$.

The *variables* X are an S-indexed family $X = \{X_s\}_{s \in S}$ of disjoint variable sets with each X_s countably infinite. Expressions $T_\Sigma(X)_s$ and $T_{\Sigma,s}$ denote, respectively, the *set of terms of sort* s and the *set of ground terms of sort* s; accordingly, $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$ denote the corresponding order-sorted Σ-term algebras. All order-sorted signatures are assumed *preregular* [19], i.e., each Σ-term t has a *least sort* $ls(t) \in S$ s.t. $t \in T_\Sigma(X)_{ls(t)}$. For $S' \subseteq S$, a term is called *S'-linear* if no variable with sort in S' occurs in it twice. The *set of variables* of t is written $vars(t)$.

A *substitution* is an S-indexed mapping $\theta : X \longrightarrow T_\Sigma(X)$ that is different from the identity only for a finite subset of X. The identity substitution is denoted by id and $\theta|_Y$ denotes the restriction of θ to a family of variables $Y \subseteq X$. Expression $dom(\theta)$ denotes the domain of θ, i.e., the subfamily of X for which $\theta(x) \neq x$, and $ran(\theta)$ denotes the family of variables introduced by $\theta(x)$, for $x \in dom(\theta)$. Substitutions extend homomorphically to terms in the natural way. A substitution θ is called *ground* iff $ran(\theta) = \emptyset$. 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 $\{x_1, \ldots, x_n\} \subseteq vars(c)$; it can be viewed as an n-ary function $C(t_1, \ldots, t_n) = c\theta$, where $\theta(x_i) = t_i$ for $1 \leq i \leq n$ and $\theta(x) = x$ otherwise.

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. We assume throughout that $T_{\Sigma,s} \neq \emptyset$ 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 $t =_\mathcal{E} u$ iff $\mathcal{E} \vdash t = u$ by the deduction rules for order-sorted equational logic in [25]. Similarly, $=^1_\mathcal{E}$ denotes provable $\mathcal{E}$-equality in *one step* of deduction. 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 $t \ll_\mathcal{E} u$ iff 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 subset of sorts $S' \subseteq S$ iff for any $t = u \in E$ and any substitution $\theta : X \longrightarrow T_\Sigma(X)$ neither $t\theta$ nor $u\theta$ are a variable for some sort $s \in S'$. $\mathcal{T}_\mathcal{E}(X)$ and $\mathcal{T}_\mathcal{E}$ (also written $\mathcal{T}_{\Sigma/E}(X)$ 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; $\mathcal{T}_{\Sigma/E}$ 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* iff 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* iff $vars(t) = vars(u)$ for any equation $(t = u \textbf{ if } \gamma) \in E$.

Appropriate requirements are needed to make an equational theory $\mathcal{E}$ *admissible*, i.e., *executable* in rewriting languages such as Maude [14]. In this paper, it is assumed that the equations of $\mathcal{E}$ can be decomposed into a disjoint union $E \uplus B$, 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, and 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. $\overrightarrow{E}$ is *sort decreasing* modulo B iff for each $(t \rightarrow u \textbf{ if } \gamma) \in \overrightarrow{E}$ and substitution θ, $ls(t\theta) \geq ls(u\theta)$ if $(\Sigma, B, \overrightarrow{E}) \vdash \gamma\theta$. $\overrightarrow{E}$ is *operationally terminating* modulo B iff there is no infinite well-formed proof tree in $(\Sigma, B, \overrightarrow{E})$. $\overrightarrow{E}$ is *confluent* modulo B iff for all $t, t_1, t_2 \in T_\Sigma(X)$, if $t \rightarrow^*_{E/B} t_1$ and $t \rightarrow^*_{E/B} t_2$, then there is $u \in T_\Sigma(X)$ such that $t_1 \rightarrow^*_{E/B} u$ and $t_2 \rightarrow^*_{E/B} u$. The term $t\downarrow_{E/B} \in T_\Sigma(X)$ denotes the *E-canonical form* of t modulo B so that $t \rightarrow^*_{E/B} t\downarrow_{E/B}$ and $t\downarrow_{E/B}$ cannot be further reduced by $\rightarrow_{E/B}$. Under the above assumptions $t\downarrow_{E/B}$ is unique up to B-equality.

A *Σ-rule* is a triple $l \rightarrow r \textbf{ if } \phi$, with $l, r \in T_\Sigma(X)_s$, for some sort $s \in S$, and $\phi = \bigwedge_{i \in I} t_i = u_i$ a finite conjunction of Σ-equations. A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E, R)$ with (Σ, E) an order-sorted equational theory and R a finite set of Σ-rules. $\mathcal{R}$ induces a rewrite relation $\rightarrow_\mathcal{R}$ on $T_\Sigma(X)$ defined for every $t, u \in T_\Sigma(X)$ by $t \rightarrow_\mathcal{R} u$ iff there is a rule $(l \rightarrow r \textbf{ if } \phi) \in R$ and a substitution $\theta : X \longrightarrow T_\Sigma(X)$ satisfying $t =_E l\theta$, $u =_E r\theta$, and $E \vdash \phi\theta$. The relation $\rightarrow_\mathcal{R}$ is undecidable in general, unless conditions such as coherence [37] are given. A key point of this paper is to make such a relation decidable when E decomposes as $\mathcal{E}_0 \uplus B_1$, where $\mathcal{E}_0$ is a built-in theory for which formula satisfiability is decidable and B_1 has a matching algorithm. A *topmost rewrite theory* is a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, such that for some top sort *State*, no operator in Σ has *State* as argument sort and each rule $l \rightarrow r \textbf{ if } \phi \in R$ satisfies $l, r \in T_\Sigma(X)_{State}$ and $l \notin X$.

3 Rewriting Modulo a Built-in Subtheory

The concept of rewriting modulo a built-in equational subtheory is presented. In particular, the notion of rewrite theory modulo a built-in subtheory and its ground rewrite relation are introduced. A canonical representation for rewrite theories modulo built-ins is proposed and fundamental results are presented. Detailed proofs can be found in [33, 34].

Definition 1 (Signature with Built-ins). *An order-sorted signature $\Sigma = (S, \leq, F)$ is a signature with* built-in subsignature *$\Sigma_0 \subseteq \Sigma$ iff $\Sigma_0 = (S_0, F_0)$ is many-sorted, S_0 is a set of minimal elements in $(S, \leq)$, and if $f : w \longrightarrow s \in F_1$, then $s \notin S_0$ and f has no other typing in F_0, where $F_1 = F \backslash F_0$.*

The notion of built-in subsignature in an order-sorted signature Σ is modeled by a many-sorted signature Σ_0 defining the built-in terms $T_{\Sigma_0}(X_0)$. The restriction imposed on the sorts and the function symbols in Σ w.r.t. Σ_0 provides a clear syntactic distinction between built-in terms (the only ones with built-in sorts) and all other terms.

If $\Sigma \supseteq \Sigma_0$ is a signature with built-ins, then an *abstraction of built-ins* for t is a context $\lambda x_1 \cdots x_n.t^\circ$ such that $t^\circ \in T_{\Sigma_1}(X)$ and $\{x_1, \ldots, x_n\} = vars(t^\circ) \cap X_0$, where $\Sigma_1 = (S, \leq, F_1)$ and $X_0 = \{X_s\}_{s \in S_0}$. Lemma 1 shows that such an abstraction can be chosen so as to provide a canonical decomposition of t with useful properties.

Lemma 1. *Let Σ be a signature with built-in subsignature $\Sigma_0 = (S_0, F_0)$. For each $t \in T_\Sigma(X)$, there exist an abstraction of built-ins $\lambda x_1 \cdots x_n.t^\circ$ for t and a substitution $\theta^\circ : X_0 \longrightarrow T_{\Sigma_0}(X_0)$ such that (i) $t = t^\circ\theta^\circ$ and (ii) $dom(\theta^\circ) = \{x_1, \ldots, x_n\}$ are pairwise distinct and disjoint from $vars(t)$; moreover, (iii) t° can always be selected to be S_0-linear and with $\{x_1, \ldots, x_n\}$ disjoint from an arbitrarily chosen finite subset Y of X_0.*

In the rest of the paper, for any $t \in T_\Sigma(X)$ and $Y \subseteq X_0$ finite, the expression $abstract_{\Sigma_1}(t, Y)$ denotes the choice of a triple $\langle \lambda x_1 \cdots x_n.t^\circ ; \theta^\circ ; \phi^\circ \rangle$ such that the context $\lambda x_1 \cdots x_n.t^\circ$ and the substitution θ° satisfy the properties (i)–(iii) in Lemma 1, and $\phi^\circ = \bigwedge_{i=1}^n (x_i = \theta^\circ(x_i))$.

Under certain restrictions on axioms, matching a Σ-term t to a Σ-term u, can be decomposed modularly into Σ_1-matching of the corresponding λ-abstraction and Σ_0-matching of the built-in subterms. This is described in Lemma 2.

Lemma 2. *Let $\Sigma = (S, \leq, F)$ be a signature with built-in subsignature $\Sigma_0 = (S_0, F_0)$. Let B_0 be a set of Σ_0-axioms and B_1 a set of Σ_1-axioms. For B_0 and B_1 regular, linear, collapse free for any sort in S_0, and sort-preserving, if $t \in T_{\Sigma_1}(X_0)$ is linear with $vars(t) = \{x_1, \ldots, x_n\}$, then for each $\theta : X_0 \longrightarrow T_{\Sigma_0}(X_0)$:*

(a) if $t\theta =^1_{B_0} t'$, then there exist $x \in \{x_1, \ldots, x_n\}$ and $w \in T_{\Sigma_0}(X_0)$ such that $\theta(x) =^1_{B_0} w$ and $t' = t\theta'$, with $\theta'(x) = w$ and $\theta'(y) = \theta(y)$ otherwise;

(b) if $t\theta =^1_{B_1} t'$, then there exists $v \in T_{\Sigma_1}(X_0)$ such that $t =^1_{B_1} v$ and $t' = v\theta$; and

(c) if $t\theta =_{B_0 \uplus B_1} t'$, then there exist $v \in T_{\Sigma_1}(X_0)$ and $\theta' : X_0 \longrightarrow T_{\Sigma_0}(X_0)$ such that $t' = v\theta'$, $t =_{B_1} v$, and $\theta =_{B_0} \theta'$ (i.e., $\theta(x) =_{B_0} \theta'(x)$ for each $x \in X_0$).

Definition 2 introduces the notion of rewriting modulo a built-in subtheory.

Definition 2 (Rewriting Modulo a Built-in Subtheory). *A* rewrite theory modulo the built-in subtheory $\mathcal{E}_0$ *is a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$ with:*

(a) Σ=$(S, \leq, F)$ a signature with built-in subsignature Σ_0=(S_0, F_0) and top sort State$\in S$;

(b) $E = E_0 \uplus B_0 \uplus B_1$, where E_0 is a set of Σ_0-equations, B_0 (resp., B_1) are Σ_0-axioms (resp., Σ_1-axioms) satisfying the conditions in Lemma 2, $\mathcal{E}_0 = (\Sigma_0, E_0 \uplus B_0)$ and $\mathcal{E} = (\Sigma, E)$ are admissible, and the theory inclusion $\mathcal{E}_0 \subseteq \mathcal{E}$ is protecting;

(c) *R is a set of* rewrite rules *of the form* $l(\overrightarrow{x_1}, \overrightarrow{y}) \rightarrow r(\overrightarrow{x_2}, \overrightarrow{y})$ ***if*** $\phi(\overrightarrow{x_3})$ *such that* $l, r \in T_\Sigma(X)_{State}$, *$l$ is* $(S \setminus S_0)$*-linear,* $\overrightarrow{x_i}{:}\overrightarrow{s_i}$ *with* $\overrightarrow{s_i} \in S_0^*$, *for* $i \in \{1,2,3\}$, $\overrightarrow{y}{:}\overrightarrow{s}$ *with* $\overrightarrow{s} \in (S \setminus S_0)^*$, *and* $\phi \in QF_{\Sigma_0}(X_0)$, *where* $QF_{\Sigma_0}(X_0)$ *denotes the set of quantifier-free* Σ_0*-formulas with variables in* X_0.

Note that no assumption is made on the relationship between the built-in variables x_1 in the left-hand side, x_2 in the right-hand side, and x_3 in the condition ϕ of a rewrite rule. This freedom is key for specifying open systems with a rewrite theory because, for instance, x_2 can have more variables than x_1. On the other hand, due to the presence of conditions ϕ in the rules of $\mathcal{R}$ that are general quantifier-free formulas, as opposed to a conjunction of atoms, properly speaking $\mathcal{R}$ is more general than a standard rewrite theory as defined in Section 2.

The binary rewrite relation induced by a rewrite theory $\mathcal{R}$ modulo $\mathcal{E}_0$ on $T_{\Sigma,State}$ is called the *ground rewrite relation* of $\mathcal{R}$.

Definition 3 (Ground Rewrite Relation). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo $\mathcal{E}_0$. The relation $\rightarrow_{\mathcal{R}}$ induced by $\mathcal{R}$ on $T_{\Sigma,State}$ is defined for $t, u \in T_{\Sigma,State}$ by $t \rightarrow_{\mathcal{R}} u$ iff there is a rule $l \rightarrow r$ **if** ϕ in R and a ground substitution $\sigma : X \longrightarrow T_\Sigma$ such that (a) $t =_E l\sigma$, $u =_E r\sigma$, and (b) $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$.*

The ground rewrite relation $\rightarrow_{\mathcal{R}}$ is the topmost rewrite relation induced by R modulo E on $T_{\Sigma,State}$. This relation is defined even when a rule in R has extra variables in its right-hand side: the rule is then non-deterministic and such extra variables can be arbitrarily instantiated, provided that the corresponding instantiation of ϕ holds. Also, note that non-built-in variables can occur in l, but $\phi\sigma$ is a *variable-free formula* in $QF_{\Sigma_0}(\emptyset)$, so that either $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$ or $\mathcal{T}_{\mathcal{E}_0} \not\models \phi\sigma$.

A rewrite theory $\mathcal{R}$ modulo $\mathcal{E}_0$ always has a canonical representation in which all left-hand sides of rules are S_0-linear Σ_1-terms.

Definition 4 (Normal Form of a Rewrite Theory Modulo $\mathcal{E}_0$). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo $\mathcal{E}_0$. Its* normal form $\mathcal{R}^\circ = (\Sigma, E, R^\circ)$ *has rules:*

$$R^\circ = \{l^\circ \rightarrow r \textbf{ if } \phi \wedge \phi^\circ \mid (\exists l \rightarrow r \textbf{ if } \phi \in R)\langle \lambda\overrightarrow{x}.l^\circ\,;\theta^\circ\,;\phi^\circ\rangle = abstract_\Sigma(l, vars(\{l, r, \phi\}))\}.$$

Lemma 3 (Invariance of Ground Rewriting under Normalization). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo $\mathcal{E}_0$. Then $\rightarrow_{\mathcal{R}} = \rightarrow_{\mathcal{R}^\circ}$.*

By the properties of the axioms in a rewrite theory modulo built-ins $\mathcal{R} = (\Sigma, E_0 \uplus B_0 \uplus B_1)$ (see Definition 2), B_1-matching a term $t \in T_\Sigma(X_0)$ to a left-hand side l° of a rule in R° provides a complete unifiability algorithm for ground B_1-unification of t and l°.

Lemma 4 (Matching Lemma). *Let $\mathcal{R} = (\Sigma, E_0 \uplus B_0 \uplus B_1, R)$ be a rewrite theory modulo $\mathcal{E}_0$. For $t \in T_\Sigma(X_0)_{State}$ and l° a left-hand side of a rule in R° with $vars(t) \cap vars(l^\circ) = \emptyset$, $t \ll_{B_1} l^\circ$ iff $GU_{B_1}(t = l^\circ) \neq \emptyset$ holds, where $GU_{B_1}(t = l^\circ) = \{\sigma : X \longrightarrow T_\Sigma \mid t\sigma =_{B_1} l^\circ\sigma\}$.*

4 Symbolic Rewriting Modulo a Built-in Subtheory

This section explains how a rewrite theory $\mathcal{R}$ modulo $\mathcal{E}_0$ defines a symbolic rewrite relation on terms in $T_{\Sigma_0}(X_0)_{State}$ constrained by formulas in $QF_{\Sigma_0}(X_0)$. The key idea is that, when $\mathcal{E}_0$ is a decidable theory, transitions on the symbolic terms can be performed by rewriting modulo B_1, and satisfiability of the formulas can be handled by an SMT decision procedure. This approach provides an efficiently executable symbolic method called *rewriting modulo SMT* that is sound and complete with respect to the ground rewrite relation of Definition 3 and yields a complete symbolic reachability analysis method. Detailed proofs of the theorems presented in this section can be found in [34].

Definition 5 (Constrained Terms and their Denotation). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo $\mathcal{E}_0$. A* constrained term *is a pair $\langle t\,;\varphi\rangle$ in $T_\Sigma(X_0)_{State} \times QF_{\Sigma_0}(X_0)$. Its* denotation $[\![t]\!]_\varphi$ *is defined as* $[\![t]\!]_\varphi = \{t' \in T_{\Sigma,State} \mid (\exists \sigma : X_0 \longrightarrow T_{\Sigma_0})\ t' = t\sigma \wedge \mathcal{T}_{\mathcal{E}_0} \models \varphi\sigma\}$.

The domain of σ in Definition 5 ranges over all built-in variables X_0 and consequently $[\![t]\!]_\varphi \subseteq T_{\Sigma,State}$ for any $t \in T_\Sigma(X_0)_{State}$, even if $vars(t) \not\subseteq vars(\varphi)$. Intuitively, $[\![t]\!]_\varphi$ denotes the set of all ground states that are instances of t and satisfy φ.

Before introducing the symbolic rewrite relation on constrained terms induced by a rewrite theory $\mathcal{R}$ modulo $\mathcal{E}_0$, auxiliary notation for variable renaming is required. In the rest of the paper, the expression *fresh-vars*(Y), for $Y \subseteq X$ finite, represents the choice of a variable renaming $\zeta : X \longrightarrow X$ satisfying $Y \cap ran(\zeta) = \emptyset$.

Definition 6 (Symbolic Rewrite Relation). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo built-ins $\mathcal{E}_0$. The* symbolic rewrite relation $\rightsquigarrow_{\mathcal{R}}$ *induced by $\mathcal{R}$ on $T_\Sigma(X_0)_{State} \times QF_{\Sigma_0}(X_0)$ is defined for $t, u \in T_\Sigma(X_0)_{State}$ and $\varphi, \varphi' \in QF_{\Sigma_0}(X_0)$ by $\langle t\,;\varphi\rangle \rightsquigarrow_{\mathcal{R}} \langle u\,;\varphi'\rangle$ iff there is a rule $l \to r$ **if** ϕ in R and a substitution $\theta : X \longrightarrow T_\Sigma(X)$ such that (a) $t =_E l\zeta\theta$ and $u = r\zeta\theta$, (b) $\mathcal{E}_0 \vdash (\varphi' \Leftrightarrow \varphi \wedge \phi\zeta\theta)$, and (c) φ' is $\mathcal{T}_{\mathcal{E}_0}$-satisfiable, where $\zeta =$ fresh-vars$(vars(t, \varphi))$.*

The symbolic relation $\rightsquigarrow_{\mathcal{R}}$ on constrained terms is defined as a topmost rewrite relation induced by R modulo E on $T_\Sigma(X_0)$ with extra bookkeeping of constraints. Note that φ' in $\langle t\,;\varphi\rangle \rightsquigarrow_{\mathcal{R}} \langle u\,;\varphi'\rangle$, when witnessed by $l \to r$ **if** ϕ and θ, is *semantically equivalent* to $\varphi \wedge \phi\zeta\theta$, in contrast to being *syntactically* equal. This extra freedom allows for simplification of constraints if desired. Also, such a constraint φ' is satisfiable in $\mathcal{T}_{\mathcal{E}_0}$, implying that φ and $\phi\theta$ are both satisfiable in $\mathcal{T}_{\mathcal{E}_0}$, and therefore $[\![t]\!]_\varphi \neq \emptyset \neq [\![u]\!]_{\varphi'}$. Note that, up to the choice of the semantically equivalent φ' for which a fixed strategy is assumed, the symbolic relation $\rightsquigarrow_{\mathcal{R}}$ is deterministic because the renaming of variables in the rules is fixed by *fresh-vars*. This is key when executing $\rightsquigarrow_{\mathcal{R}}$, as explained in Section 5.

The important question to ask is whether this symbolic relation soundly and completely simulates its ground counterpart. The rest of this section answers this question in the affirmative for *normalized* rewrite theories modulo built-ins. Thanks to Lemma 3, the

conclusion is therefore that $\rightsquigarrow_{\mathcal{R}^\circ}$ soundly and completely simulates $\rightarrow_{\mathcal{R}}$ for any rewrite theory $\mathcal{R}$ modulo built-ins $\mathcal{E}_0$.

The soundness of $\rightsquigarrow_{\mathcal{R}^\circ}$ w.r.t. $\rightarrow_{\mathcal{R}^\circ}$ is stated in Theorem 1.

Theorem 1 (Soundness). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo built-ins $\mathcal{E}_0$, $t, u \in T_\Sigma(X_0)_{State}$, and $\varphi, \varphi' \in QF_{\Sigma_0}(X_0)$. If $\langle t\,;\varphi\rangle \rightsquigarrow_{\mathcal{R}^\circ} \langle u\,;\varphi'\rangle$, then $t\rho \rightarrow_{\mathcal{R}^\circ} u\rho$ for all $\rho : X_0 \longrightarrow T_{\Sigma_0}$ satisfying $\mathcal{T}_{\mathcal{E}_0} \models \varphi'\rho$.*

The completeness of $\rightsquigarrow_{\mathcal{R}^\circ}$ w.r.t. $\rightarrow_{\mathcal{R}^\circ}$ is stated in Theorem 2, which is a "lifting lemma". Intuitively, completeness states that a symbolic relation yields an over-approximation of its ground rewriting counterpart.

Theorem 2 (Completeness). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo built-ins $\mathcal{E}_0$, $t \in T_\Sigma(X_0)_{State}$, $u' \in T_{\Sigma,State}$, and $\varphi \in QF_{\Sigma_0}(X_0)$. For any $\rho : X_0 \longrightarrow T_{\Sigma_0}$ such that $t\rho \in [\![t]\!]_\varphi$ and $t\rho \rightarrow_{\mathcal{R}^\circ} u'$, there exist $u \in T_\Sigma(X_0)_{State}$ and $\varphi' \in QF_{\Sigma_0}(X_0)$ such that $\langle t\,;\varphi\rangle \rightsquigarrow_{\mathcal{R}^\circ} \langle u\,;\varphi'\rangle$ and $u' \in [\![u]\!]_{\varphi'}$.*

Although the above soundness and completeness theorems, plus Lemma 3, show that $\rightarrow_{\mathcal{R}}$ is characterized symbolically by $\rightsquigarrow_{\mathcal{R}^\circ}$, for any rewrite theory $\mathcal{R}$ modulo $\mathcal{E}_0$, because of Condition (c) in Definition 6, the relation $\rightsquigarrow_{\mathcal{R}^\circ}$ is in general undecidable. However, $\rightsquigarrow_{\mathcal{R}^\circ}$ becomes decidable for built-in theories $\mathcal{E}_0$ that can be extended to a *decidable theory* $\mathcal{E}_0^+$ (typically by adding some inductive consequences) such that:

$$(\forall \phi \in QF_{\Sigma_0}(X_0))\ \phi \text{ is } \mathcal{E}_0^+\text{-satisfiable} \iff (\exists \sigma : X_0 \longrightarrow T_{\Sigma_0})\ \mathcal{T}_{\mathcal{E}_0} \models \phi\sigma. \tag{1}$$

Many decidable theories $\mathcal{E}_0^+$ of interest are supported by SMT solvers satisfying this requirement. For example, $\mathcal{E}_0$ can be the equational theory of natural number addition and $\mathcal{E}_0^+$ Pressburger arithmetic. That is, $\mathcal{T}_{\mathcal{E}_0}$ is the *standard model* of both $\mathcal{E}_0$ and $\mathcal{E}_0^+$, and $\mathcal{E}_0^+$-satisfiability coincides with satisfiability in such a standard model. Under such conditions, satisfiability of $\varphi \wedge \phi\zeta\theta$ (and therefore of φ') in a step $\langle t\,;\varphi\rangle \rightsquigarrow_{\mathcal{R}^\circ} \langle u\,;\varphi'\rangle$ becomes decidable by invoking an SMT-solver for $\mathcal{E}_0$, so that $\rightsquigarrow_{\mathcal{R}^\circ}$ can be naturally described as *symbolic rewriting modulo SMT* (and modulo B_1).

The symbolic reachability problems considered for a rewrite theory $\mathcal{R}$ modulo $\mathcal{E}_0$ in this paper, are existential formulas of the form $(\exists \overrightarrow{z})\ t \rightarrow^* u \wedge \varphi$, with $\overrightarrow{z}$ the variables appearing in t, u, and φ, $t \in T_\Sigma(X_0)_{State}$, $u \in T_\Sigma(X)_{State}$, and $\varphi \in QF_{\Sigma_0}(X_0)$. By abstracting the Σ_0-subterms of u, the ground solutions of such a reachability problem are those witnessing the model-theoretic satisfaction relation:

$$\mathcal{T}_{\mathcal{R}} \models (\exists \overrightarrow{x} \uplus \overrightarrow{y})\ t(\overrightarrow{x}) \rightarrow^* u^\circ(\overrightarrow{y}) \wedge \varphi_1(\overrightarrow{x}) \wedge \varphi_2(\overrightarrow{x}, \overrightarrow{y}) \tag{2}$$

where $\mathcal{T}_{\mathcal{R}} = (\mathcal{T}_{\Sigma/E}, \rightarrow_{\mathcal{R}}^*)$ is the initial reachability model of $\mathcal{R}$ [12], $t \in T_\Sigma(X_0)$ and $u^\circ \in T_{\Sigma_1}(X)$ are S_0-linear, $vars(t) \subseteq \overrightarrow{x} \subseteq X_0$, and $\overrightarrow{y} \subseteq X$. Thanks to the soundness and completeness results, Theorem 1 and Theorem 2, the solvability of Condition (b) for $\rightarrow_{\mathcal{R}}$ can be achieved by reachability analysis with $\rightsquigarrow_{\mathcal{R}^\circ}$, as stated in Theorem 3.

Theorem 3 (Symbolic Reachability Analysis). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory modulo built-ins $\mathcal{E}_0$. The model-theoretic satisfaction relation in (2) has a solution iff there exist a term $v \in T_\Sigma(X)_{State}$, a constraint $\varphi' \in QF_{\Sigma_0}(X_0)$, and a substitution $\theta : X \longrightarrow T_\Sigma(X)$, with $dom(\theta) \subseteq \overrightarrow{y}$, such that (a) $\langle t\,;\varphi_1\rangle \rightsquigarrow^*_{\mathcal{R}^\circ} \langle v\,;\varphi'\rangle$, (b) $v =_{B_1} u^\circ\theta$, and (c) $\varphi' \wedge \varphi_2\theta$ is $\mathcal{T}_{\mathcal{E}_0}$-satisfiable.*

In Theorem 3, since $dom(\theta) \subseteq \overrightarrow{y}$, and $\overrightarrow{x}$ and $\overrightarrow{y}$ are disjoint, the variables of $\overrightarrow{x}$ in $\varphi_2\theta$ are left unchanged. Therefore, $\varphi_2\theta$ *links* the requirements for the variables $\overrightarrow{x}$ in the initial state and $\overrightarrow{y}$ in the final state according to both φ_1 and φ_2. Also note that the inclusion of formula φ_1 as a conjunct in the formula in Condition (c) of Theorem 3 is superfluous because $\langle t\,;\varphi_1\rangle \rightsquigarrow_{\mathcal{R}^\circ} \langle v\,;\varphi'\rangle$ implies that φ_1 is a semantic consequence of φ'.

5 Reflective Implementation of $\rightsquigarrow_{\mathcal{R}^\circ}$

The design and implementation of a prototype that offers support for symbolic rewriting modulo SMT in the Maude system are discussed. The prototype relies on Maude's meta-level features, that implement rewriting logic's reflective capabilities, and on SMT solving for $\mathcal{E}_0^+$ integrated in Maude as CVC3's decision procedures. The extension of Maude with CVC3 is available from the Matching Logic Project [35]. In the rest of this section, $\mathcal{R} = (\Sigma, E_0 \uplus B_0 \uplus B_1, R)$ is a rewrite theory modulo built-ins $\mathcal{E}_0$, where $\mathcal{E}_0$ satisfies Condition (1) in Section 4. The theory mapping $\mathcal{R} \mapsto \mathbf{u}(\mathcal{R})$ removes the constraints from the rules in R.

In Maude, reflection is efficiently supported by its *META-LEVEL* module [14], which provides key functionality for rewriting logic's *universal theory* $\mathcal{U}$ [15]. In particular, rewrite theories $\mathcal{R}$ are meta-represented in $\mathcal{U}$ as terms $\overline{\mathcal{R}}$ of sort *Module*, and a term t in $\mathcal{R}$ is meta-represented in $\mathcal{U}$ as a term $\bar{t}$ of sort *Term*. The key idea of the reflective implementation is to reduce symbolic rewriting with $\rightsquigarrow_{\mathcal{R}^\circ}$ to *standard rewriting* in an associated reflective rewrite theory extending the universal theory $\mathcal{U}$. This is specially important for formal analysis purposes, because it makes available to $\rightsquigarrow_{\mathcal{R}^\circ}$ some formal analysis features provided by Maude for rewrite theories such as reachability analysis by search. This is illustrated by the case study in Section 6.

The prototype defines a parametrized functional module $SAT(\Sigma_0, E_0 \uplus B_0)$ of quantifier-free formulas with Σ_0-equations as atoms. In particular, this module extends $(\Sigma_0, E_0 \uplus B_0)$ with new sorts *Atom* and *QFFormula*, and new *constants* $var(X_0)$ identifying the variables X_0. It has, among other functions, a function $sat : QFFormula \longrightarrow Bool$ such that for ϕ, $sat(\phi) = \top$ if ϕ is $\mathcal{E}_0^+$-satisfiable, and $sat(\phi) = \bot$ otherwise.

The process of computing the one-step rewrites of a given constrained term $\langle t\,;\varphi\rangle$ under $\rightsquigarrow_{\mathcal{R}^\circ}$ is decomposed into two conceptual steps using Maude's metalevel. First, all possible triples $\langle u\,;\theta\,;\phi\rangle$ such that $t \rightarrow_{\mathbf{u}(\mathcal{R}^\circ)} u$ is witnessed by a matching substitution θ and a rule with constraint ϕ are computed[1]. Second, these triples are filtered out by keeping only those for which the quantifier-free formula $\varphi \wedge \phi\theta$ is $\mathcal{E}_0^+$-satisfiable.

[1] Note that in $\mathbf{u}(\mathcal{R}^\circ)$ variables in X_0 are interpreted as *constants*. Therefore, the number of matching substitutions θ thus obtained is finite.

The first step in the process is mechanized by function *next*, available from the parametrized module $NEXT(\overline{\mathcal{R}}, \overline{State}, \overline{QFFormula})$ where $\overline{\mathcal{R}}$, $\overline{State}$, and $\overline{QFFormula}$ are the metalevel representations, respectively, of the rewrite theory module $\mathcal{R}$, the state sort *State*, and the quantifier-free formula sort *QFFormula*. Function *next* uses Maude's *meta-match* function and the auxiliary function *new-vars* for computing fresh variables (see Section 4). In particular, the call $next(\overline{((S, \leq, F \uplus var(X_0)), E_0 \uplus B_0 \uplus B_1, R^\circ)}, \overline{t}, \overline{\varphi})$ computes all possible triples $\langle \overline{u} \,;\, \overline{\theta'} \,;\, \overline{\phi'} \rangle$ such that $t \rightsquigarrow_{\mathcal{R}^\circ} u$ is witnessed by a substitution θ' and a rule with constraint ϕ'. More precisely, such a call first computes a renaming $\zeta = fresh\text{-}vars(vars(t, \varphi))$ and then, for each rule$(l^\circ \to r \textbf{ if } \phi)$, it uses the function *meta-match* to obtain a substitution $\overline{\theta} \in \overline{meta\text{-}match((S, \leq, F \uplus var(X_0)), B_0 \uplus B_1), t\downarrow_{E_0/B_0 \uplus B_1}, l^\circ\zeta)}$, and returns $\langle \overline{u} \,;\, \overline{\theta'} \,;\, \overline{\phi'} \rangle$ with $\overline{u} = \overline{r\zeta\theta}$, $\overline{\theta'} = \overline{\zeta\theta}$, and $\overline{\phi'} = \overline{\phi\zeta\theta}$. Note that by having a *deterministic* choice of fresh variables (including those in the constraint), function *next* is actually a *deterministic* function.

Using the above-mentioned infrastructure, the parametrized module *NEXT* implements the symbolic rewrite relation $\rightsquigarrow_{\mathcal{R}^\circ}$ as a *standard rewrite relation* in the theory *NEXT*, extending *META-LEVEL*, by means of the following conditional rewrite rule:

$$\textbf{ceq} \quad \langle X{:}State \,;\, \varphi{:}QFFormula \rangle \to \langle Y{:}State \,;\, \varphi'{:}QFFormula \rangle$$
$$\textbf{if} \quad \langle \overline{Y} \,;\, \overline{\theta} \,;\, \overline{\phi} \rangle \; S := next(\overline{\mathcal{R}^\bullet}, \overline{X}, \overline{\varphi}) \wedge sat(\varphi \wedge \phi) = \top \wedge \varphi' := \varphi \wedge \phi$$

where $\mathcal{R}^\bullet = ((S, \leq, F \uplus var(X_0)), B, R^\circ)$. Therefore, a call to an external SMT solver is just an invocation of the function *sat* in $SAT(\Sigma_0, E_0 \uplus B_0)$ in order to achieve the above functionality more efficiently and in a built-in way.

Given that the symbolic rewrite relation $\rightsquigarrow_{\mathcal{R}^\circ}$ is encoded as a standard rewrite relation, symbolic search can be *directly implemented* in Maude by its *search* command. In particular, for terms t, u°, constraints φ_1, φ_2, F a variable of sort *QFFormula*, the following invocation solves the inductive reachability problem in Condition (2):

$$search \;\; \langle t \,;\, \varphi_1 \rangle \to^* \langle u^\circ \,;\, F \rangle \;\; such\ that \;\; sat(F \wedge \varphi_2).$$

6 Analysis of the CASH algorithm

This section presents an example, developed jointly with Kyungmin Bae, of a real-time system that can be symbolically analyzed in the prototype tool described in Section 5. The analysis applies model checking based on *rewriting modulo SMT*. Some details are omitted. Full details and the prototype tool can be found in [9].

The example involves the symbolic analysis of the CASH scheduling algorithm [13], which attempts to maximize system performance while guaranteeing that critical tasks are executed in a timely manner. This is achieved by maintaining a queue of unused execution budgets that can be reused by other jobs to maximize processor utilization. CASH poses non-trivial modeling and analysis challenges because it contains an unbounded queue. Unbounded data types cannot be modeled in timed-automata formalisms, such as those of UPPAAL [22] or Kronos [39], which assume a finite discrete state.

The CASH algorithm was specified and analyzed in Real-Time Maude by *explicit-state model checking* in an earlier paper by Ölveczky and Caccamo [30], which showed that, under certain variations on both the assumptions and the design of the protocol, it could miss deadlines. But explicit-state model checking has intrinsic limitations which the new analysis by rewriting modulo SMT presented below overcomes. The CASH algorithm is parametrized by: (i) the number N of servers in the system, and (ii) the values of a maximum budget b_i and period p_i, for each server $1 \leq i \leq N$. Even if N is fixed, *there are infinitely many initial states* for N servers, since the maximum budgets b_i and periods p_i range over the natural numbers. Therefore, explicit state model checking cannot perform a full analysis. If a counterexample for N servers exists, it may be found by explicit-state model checking for some chosen initial states, as done in [31], but it could be missed if the wrong initial states are chosen.

Rewriting modulo SMT is useful for symbolically analyzing infinite-state systems like CASH. Infinite sets of states are symbolically described by terms which may involve user-definable data structures such as queues, but whose only variables range over decidable types for which an SMT solving procedure is available. For the CASH algorithm, the built-in data types used are the Booleans (sort `iBool`) and the integers (sort `iInt`). Integer built-in terms are used to model discrete time. Boolean built-in terms are used to impose constraints on integers.

A symbolic state is a pair `{iB,Cnf}` of sort `Sys` consisting of a Boolean constraint `iB`, with *and* denoted `^`, and a multiset configuration of objects `Cnf`, with multiset union denoted by juxtaposition, where each object is a record like-structure with an object identifier, a class name, and a set of attribute-value pairs. In each object configuration there is a global object (of class `global`) that models the time of the system (with attribute name `time`), the priority queue (with attribute name `cq`), the availability (with attribute name `available`), and a deadline missed flag (with attribute name `deadline-miss`). A configuration can also contain any number of server objects (of class `server`). Each server object models the maximum budget (the maximum time within which a given job will be finished, with attribute name `maxBudget`), period (with attribute name `period`), internal state (with attribute name `state`), time executed (with attribute name `timeExecuted`), budget time used (with attribute name `usedOfBudget`), and time to deadline (with attribute name `timeToDeadline`). The symbolic transitions of CASH are specified by 14 conditional rewrite rules whose conditions specify constraints solvable by the SMT decision procedure. For example, rule `[deadlineMiss]` below models the detection of a deadline miss for a server with nonzero maximum budget.

```
vars AtSG AtS : AttributeSet .  var iB : iBool .         var Cnf : Configuration .
vars iT iT' iNZT : iInt .       var St : ServerState .   vars G S : Oid .  var B : Bool .

crl [deadlineMiss] :
    { iB, < G : global | dead-miss |-> B, AtSG >
          < S : server | state |-> St, usedOfBudget |-> iT, timeToDeadline |-> iT',
                         maxBudget |-> iNZT, AtS > Cnf }
=> {iB ^ iT >= c(0) ^ iNZT > c(0) ^ iT' >  c(0) ^ iNZT > iT + iT',
          < G : global | dead-miss |-> true, AtSG >
          < S : server | state |-> St, usedOfBudget |-> iT, timeToDeadline |-> iT',
                         maxBudget |-> iNZT, AtS > Cnf }
if St =/= idle /\ check-sat(iB ^ iT >= c(0) ^ iNZT > c(0) ^ iT' >  c(0) ^ iNZT > iT + iT') .
```

That is, the protocol misses a deadline for server S whenever the value of attribute maxBudget exceeds the addition of values for usedOfBudget and timeToDeadline (i.e., iNZT > iT + iT') , so that the allocated execution time cannot be exhausted before the server's deadline.

The goal is to verify *symbolically* the existence of missed deadlines of the CASH algorithm for the *infinite set of initial configurations* containing two server objects s_0 and s_1 with maximum budgets b_0 and b_1 and periods p_0 and p_1 as unspecified natural numbers, and such that each server's maximum budget is strictly smaller than its period (i.e., $0 \leq b_0 < p_0 \wedge 0 \leq b_1 < p_1$). This infinite set of initial states is specified symbolically by the equational definition (not shown) of term symbinit. Maude's search command can then be used to symbolically check if there is a reachable state for any ground instance of symbinit that misses the deadline:

```
search in SYMBOLIC-FAILURE : symbinit =>*
  { iB:iBool,  Cnf:Configuration < g : global | AtS:AttributeSet, deadline-miss |-> true > } .
Solution 1 (state 233)
states: 234  rewrites: 60517 in 2865ms cpu (2865ms real) (21118 rewrites/second)
iB:iBool --> ((i(0) <= c(0) ^ i(1) <= c(0)) v i(0) <= c(0) + i(1) ^ ...
Cnf:Configuration -->
< s1 : server | maxBudget |-> i(0), period |-> i(1), state |-> waiting, usedOfBudget |-> c(0),
                timeToDeadline |-> ((i(1) -- c(1)) -- c(1)), timeExecuted |-> c(0) >
< s2 : server | maxBudget |-> i(2), period |-> i(3), state |-> executing, usedOfBudget |-> c(2),
                timeToDeadline |-> ((i(3) -- c(1)) -- c(1)), timeExecuted |-> c(2) >
AtS:AttributeSet --> time |-> c(2), cq |-> emptyQueue, available |-> false
```

A counterexample is found at (modeling) time two, after exploring 233 symbolic states in less than 3 seconds. By using a satisfiability witness of the constraint iB computed by the search command, a concrete counterexample is found by exploring only 54 ground states. This result compares favorably, in both time and computational resources, with the ground counterexample found by explicit-state model checking in [30], where more that 52,000 concrete states were explored before finding a counterexample.

7 Related Work and Concluding Remarks

The idea of combining term rewriting/narrowing techniques and constrained data structures is an active area of research, specially since the advent of modern theorem provers with highly efficient decision procedures in the form of SMT solvers. The overall aim of these techniques is to advance applicability of methods in symbolic verification where the constraints are expressed in some logic that has an efficient decision procedure. In particular, the work presented here has strong similarities with the narrowing-based symbolic analysis of rewrite theories initiated in [26] and extended in [8]. The main difference is the replacement of narrowing by SMT solving and the decidability advantages of SMT for constraint solving.

M. Ayala-Rincón [5] 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 also investigated. S. Falke and D. Kapur [16] studied the problem of termination of rewriting with constrained built-ins. In particular, they extended the dependency pairs framework to handle termination of equational specifications with semantic data structures and evaluation strategies in the Maude functional sublanguage. The same authors used the idea of combining rewriting induction and linear arithmetic over constrained terms [17]. Their aim is to obtain equational decision procedures that can handle semantic data types represented by the constrained built-ins. H. Kirchner and C. Ringeissen proposed the notion of constrained rewriting and have used it by combining symbolic constraint solvers [20]. The main difference between their work and rewriting modulo SMT presented in this paper, is that the former uses narrowing for symbolic execution, both at the symbolic 'pattern matching' and the constraint solving levels. In contrast, rewriting modulo SMT solves the symbolic pattern matching task by rewriting while constraint solving is delegated to an SMT decision procedure. More recently, C. Kop and N. Nishida [21] have proposed a way to unify the ideas regarding equational rewriting with logical constraints. More generally, while the approaches in [5, 16, 17, 20, 21] address symbolic reasoning for *equational* theorem proving purposes, none of them addresses the kind of non-deterministic rewrite rules, which are needed for open system modeling. More recently, A. Arusoaie et al. [4] have proposed a language-independent symbolic execution framework, within the *K* framework [23], for languages endowed with a formal operational semantics based on term rewriting. There, the built-in subtheories are the datatypes of a programming language and symbolic analysis is performed on constrained terms (called "patterns"); unification is also implemented by matching for a restricted class of rewrite rules and uses SMT solvers to check constraints.

This paper has presented rewrite theories modulo built-ins and has shown how they can be used for *symbolically* modeling and analyzing concurrent open systems, where non-deterministic values from the environment can be represented by built-in terms [33,34]. In particular, the main contributions of this paper can be summarized as follows: (1) it presents rewriting modulo SMT as a new symbolic technique combining the powers of rewriting, SMT solving, and model checking; (2) this combined power can be applied to model and analyze systems outside the scope of each individual technique; (3) in particular, it is ideally suited to model and analyze the challenging case of *open systems*; and (4) because of its reflective reduction to standard rewriting, current algorithms and tools for model checking closed systems can be *reused* in this new symbolic setting without requiring any changes to their implementation.

Under reasonable assumptions, including decidability of $\mathcal{E}_0^+$, a rewrite theory modulo is executable by term rewriting modulo SMT. This feature makes it possible to use, for symbolic analysis, state-of-the-art tools already available for Maude, such as its space search commands, with no change whatsoever required to use such tools. We have proved that the symbolic rewrite relation is sound and complete with respect to its ground counterpart, have presented an overview of the prototype that offers support for rewriting modulo SMT in Maude, and have presented a case study on the symbolic analysis of the CASH scheduling algorithm illustrating the use of these techniques.

Future work on a mature implementation and on extending the idea of rewriting modulo SMT with other symbolic constraint solving techniques such as narrowing modulo

should be pursued. Also, the extension to symbolic LTL model checking, together with state space reduction techniques, should be investigated. The ideas presented here extend results in [33] and have been successfully applied to the symbolic analysis of NASA's PLEXIL language to program open cyber-physical systems [33]. Future applications to PLEXIL and other languages should also be pursued.

References

1. E. Althaus, E. Kruglov, and C. Weidenbach. Superposition modulo linear arithmetic SUP(LA). In S. Ghilardi and R. Sebastiani, editors, *FroCos*, volume 5749 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2009.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Model Checking Software*, pages 146–162, 2006.
4. A. Arusoaie, D. Lucanu, and V. Rusu. A generic framework for symbolic execution. In M. Erwig, R. F. Paige, and E. V. Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 281–301. Springer, 2013.
5. M. Ayala-Rincón. *Expressiveness of Conditional Equational Systems with Built-in Predicates*. PhD thesis, Universität Kaiserslauten, 1993.
6. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
7. F. Baader and K. Schulz. Unification in the union of disjoint equational theories: combining decision procedures. *Journal of Symbolic Computation*, 21:211–243, 1996.
8. K. Bae, S. Escobar, and J. Meseguer. Abstract logical model checking of infinite-state systems using narrowing. In F. van Raamsdonk, editor, *RTA*, volume 21 of *LIPIcs*, pages 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.
9. K. Bae and C. Rocha. A note on symbolic reachability analysis modulo integer constraints for the CASH algorithm. Available at: `http://maude.cs.uiuc.edu/cases/scash`, 2012.
10. M. P. Bonacina, C. Lynch, and L. M. de Moura. On deciding satisfiability by theorem proving with speculative inferences. *Journal of Automated Reasoning*, 47(2):161–189, 2011.
11. A. Boudet. Combining unification algorithms. *J. Symb. Comp.*, 16(6):597–626, 1993.
12. R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.
13. M. Caccamo, G. C. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *IEEE Real-Time Systems Symposium*, pages 295–304. IEEE Computer Society, 2000.
14. 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 *LNCS*. Springer, 2007.
15. M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.
16. S. Falke and D. Kapur. Operational termination of conditional rewriting with built-in numbers and semantic data structures. *ENTCS*, 237:75–90, 2009.
17. 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.
18. M. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *ICCAD*, pages 794–801. ACM, 2006.

19. 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.
20. H. Kirchner and C. Ringeissen. Combining symbolic constraint solvers on algebraic domains. *Journal of Symbolic Computation*, 18(2):113–155, 1994.
21. C. Kop and N. Nishida. Term rewriting with logical constraints. In *FroCos*, volume 8152 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2013.
22. K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997.
23. D. Lucanu, T.-F. Serbanuta, and G. Rosu. K framework distilled. In *Rewriting Logic and Its Applications - 9th International Workshop, WRLA 2012, Held as a Satellite Event of ETAPS, Tallinn, Estonia, March 24-25, 2012, Revised Selected Papers*, volume 7571 of *Lecture Notes in Computer Science*, pages 31–53. Springer, 2012.
24. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
25. 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.
26. J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.
27. A. Milicevic and H. Kugler. Model checking using SMT and theory of lists. *NASA Formal Methods*, pages 282–297, 2011.
28. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
29. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(t). *Journal of the ACM*, 53(6):937–977, 2006.
30. P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering (FASE'06)*, volume 3922 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2006.
31. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.
32. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.
33. C. Rocha. *Symbolic Reachability Analysis for Rewrite Theories*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.
34. C. Rocha, J. Meseguer, and C. Muñoz. Rewriting modulo SMT. Technical Memorandum NASA/TM-2013-218033, NASA, Langley Research Center, Hampton VA 23681-2199, USA, August 2013.
35. 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.
36. M. Veanes, N. Bjørner, and A. Raschke. An SMT approach to bounded reachability analysis of model programs. In *FORTE*, pages 53–68. Springer, 2008.
37. P. Viry. Equational rules for rewriting logic. *TCS*, 285:487–517, 2002.
38. D. Walter, S. Little, and C. Myers. Bounded model checking of analog and mixed-signal circuits using an SMT solver. *Automated Technology for Verification and Analysis*, pages 66–81, 2007.
39. S. Yovine. Kronos: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.

Formal Specification of Button-Related Fault-Tolerance Micropatterns

Mu Sun José Meseguer musun(at)illinois.edu
meseguer(at)illinois.edu

University of Illinois at Urbana-Champaign
Illinois, USA

Abstract. Fault tolerance has been a major concern in the design of computing platforms. However, currently, fault tolerance has been done mostly with just heuristics, high level probabilistic analysis and extensive testing. In this work, we explore how we can use formal patterns to achieve fault-tolerance designs and methods. In particular, we look at faults that occur in mechanical button interfaces such as button bounce, button stuck, and phantom button faults. Our primary goal is the safety of such interfaces for medical devices [7], but the methods are more widely applicable. We formally describe corresponding patterns to address these faults including button debouncing, button stuck detection, and phantom press filtering. We prove stuttering-bisimulation results for some patterns showing their fault-masking capablities. Furthermore, for patterns where fault-masking is not possible, we prove fault-detection properties. We also instantiate these patterns to a simple instance of a button-press counter and perform execution and model checking as further validation.

1 Introduction

Idealized abstractions of computing systems allow us to build more complex applications and for more complex scenarios. One can think in terms of binary values instead of continuous voltages, and in terms of objects and messages instead of assembly-level instructions. Given the complexities of the real world, it is remarkable how accurate these abstractions can be. However, sometimes the real world behavior violates the expectation of idealized models and we refer to this type of behavior as faults.

In order to maintain the behavior of ideal models in the presence of faults, fault tolerance techniques are essential. We would like faults to be completely contained within the lower levels of design and never be exposed to the upper layers; this is the notion of *fault masking*. However, there are many cases where fault masking is impossible. In these cases, faults will inevitably be exposed to the upper layers, either by explicit fault detection or as behavioral anomalies such as extra delays and nondeterminism.

In this paper, we explore *fault-tolerance micropatterns* for button related faults including button bounce, phantom button presses, and stuck buttons. These micropatterns provide specific levels of safety for medical device interfaces in the presence of faults [7], and can be likewise applied to devices in other areas. All of these faults and fault-tolerance patterns are quite well known, but our contribution is in the formalization of these fault-tolerance models including:

(1) defining a model for button interfaces;
(2) modeling faults as a relation from ideal environments to faulty environments;
(3) describing fault tolerance methods as a design transformation pattern using parameterized modules;
(4) proving fault-tolerance results about our models using appropriate bisimulation relations; and
(5) validating of our models with execution and model checking.

Since we are dealing with faults on the interface, we mainly focus on faults in the environment. There are also other classes of faults such as internal faults (e.g. bit flips, memory corruption, computation errors). However, environmental faults and internal faults are generally handled orthogonally in the design of a system, so we focus only on environmental faults. The fault tolerance patterns that we describe in this paper all have a similar structure that is captured in Figure 1. All fault tolerance designs have a goal, an ideal abstraction that it is trying to provide (left-hand side of Figure 1). An ideal environment, and the ideal design will give the correct behavior of the system. However, the challenge comes when we have a faulty environment (right-hand side of Figure 1). Just using an ideal design with a faulty environment will most likely lead to undesirable deviations in the behavior of the system. The goal then is to provide a design transformation for the system along with the fault model that will have behavior similar to the ideal. The notion of *correspondence in behavior* is an important one. In this paper, this correspondence is expressed as a *bisimulation*.

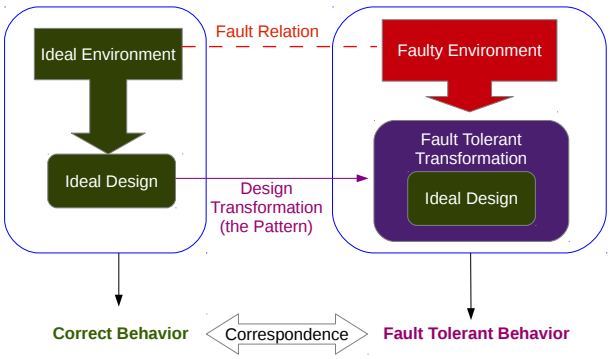


Fig. 1. Fault Modeling

The rest of the paper is organized as follows. Section 2 covers the basics of rewriting logic and the subset of Maude that we use to describe our models. Section 3 describes how we model buttons in order to describe button-related faults. Sections 4, 5, and 6 describe in detail our patterns to handel button bounce, phantom button presses, and stuck buttons respectively. We conclude in Section 7 with a summary and a discussion of potential future work.

2 Background on Parameterized Formal Specifications and Real-Time Maude

We use the Maude rewriting logic language [2] to define formal specifications for our fault-tolerance wrappers for medical systems. We present some of the basic concepts behind rewriting logic, its real-time extensions, and parametrization.

2.1 Membership Equational Logic and Rewriting Logic

Membership equational logic (MEL) [5] describes the most general form of the equational components of a Maude rewrite theory. These are called functional modules in Maude [2].

A MEL signature is a tuple (K, F, S) where S is a set of sorts (i.e. types), K is a set of kinds (i.e. super types or error types for data), and F is a set of typed function symbols (and constants). A MEL theory is a pair (Σ, E) where Σ is a MEL signature, and E a set of sentences (equations and memberships) expressing (possibly conditional) membership or equality constraints. If an MEL theory is convergent (satisfies properties of confluence, termination, and sort-decreasingness), Maude provides efficient execution of its initial model semantics.

Rewriting logic [1] describes the most general form of modules defined in Maude. A rewrite theory in Maude is defined in the form of a tuple: (Σ, E, ϕ, R), where (Σ, E) is an underlying MEL theory, ϕ defines the frozen positions of operators (positions where no rewrites are allowed to occur below), and R is a set of rewrite sentences (possibly conditional on equality and membership sentences). If a rewrite theory satisfies the properties of coherence, and the underlying MEL theory of a rewrite theory is convergent, then Maude provides efficient execution of the initial model semantics for the rewrite theory. This includes efficient execution for simulation, searching and LTL model checking.

2.2 Full Maude and Real-Time Maude

Full Maude [3] is a Maude interpreter written in Maude, which in addition to the Core Maude constructs provides syntactic constructs such as object oriented modules. Object oriented modules implicitly add in sorts `Object` and `Msg`. Furthermore, OO-modules add a sort called `Configuration` which consists of a multiset of terms of sort `Object` or `Msg`. Objects are represented as records:

```
< objectID : classID | Attribute 1 : Value 1, ... Attribute n : Value n >
```

Rewriting logic rules are then used to describe state transitions of objects based on consumption of messages. For example, the following rule expresses the fact that a surgical-laser object consumes a message to set the power to 50 Watts:

```
rl setPower(sl1, 50) < sl1 : SurgeryLaser | power : P >
=> < sl1 : SurgeryLaser | power : 50 > .
```

Real-time Maude [6] is a real-time extension for Maude developed on top of Full Maude. It adds syntactic constructs for defining timed modules. Timed modules automatically import the `TIME` module, which defines the sort `Time` (which can be instantiated as discrete or continuous) along with various arithmetic and comparison operations on `Time`. Timed modules also provide a sort `System` which encapsulates a `Configuration` and implicitly associates with it a time stamp of sort `Time`. After defining a time-advancing strategy, Real-time Maude provides timed execution (`trew`), timed search (`tsearch`), which performs search on a term of sort `System` based on the time advancement strategy, and timed and untimed LTL model checking commands.

Real-time Maude provides useful constructs for specifying real-time systems, including basic semantics of time and time advancement. We use the model of linear time provided by Real-Time Maude. For time advancement, we have used the conventional best practice where only one timed rewrite rule is used and is fully determined by the operators *tick* and *mte* [6].

The *tick* operator advances time over a configuration by some time duration. For example, with timer (and time units being seconds): $tick(timer(10), 3) = timer(7)$. That is, a timer with 10 sec remaining ticked by 3 sec will become a timer with 7 sec remaining.

The *mte* operator computes the maximum time that can elapse in a system before an interesting event occurs. Interesting events include all state transitions in which messages are generated in a configuration. Again, with the timer example, we assume that components only react when the timers expire, so the maximum time elapsable for a timer would be the time it takes the timer to expire: $mte(timer(10)) = 10$.

Real-Time Maude also includes models of time that have infinity, `INF`, as a possible time value. Although, `INF` will never be used to advance time in any system, it is useful to have `INF` to describe unbounded time. For example, $mte(\texttt{stableSys}) = \texttt{INF}$.

2.3 Parameterized Modules

Modules in Maude have an *initial model semantics*. Maude also supports *theories* which have a *loose semantics* (that is, not just the initial mode, but all the models of the theory are allowed). Theories can be instantiated by *views* (i.e., theory interpretations) to other theories or modules. In particular, a theory can be instantiated by a view to any module whose initial model satisfies all equational, membership, and rewrite sentences of the theory.

Parametrized modules [2] are modules which take theories as input parameters and define operations (parametrically) in terms of the input theory. Parametrized modules are instantiated by providing views to concrete modules for the corresponding input theories. Once instantiated, the parametrized module is given the free extension semantics for the initial models of the targets of the input views. Core Maude, Full Maude, and Real-Time Maude all support parameterized modules. For our pattern, we will exploit in particular the Real-Time Maude parameterization mechanisms.

3 Modeling Buttons

Before we describe specific patterns, we should describe the problem domain that we are addressing. Many cyber-physical systems, including many medical devices, use but-

tons as an input interface. We need a general abstraction that can capture the important details of any button interaction with the system. This abstraction must be detailed enough to model faulty button behavior.

For the cases that we are considering, it is sufficient to use a 2-state button abstraction. A button model can be in one of two states, either *pressed* or *not pressed*, at any instant in time. Button behavior is then a function $button_{state} : Time \to \{on, off\}$. Here, $Time$ is some ideal continuous physical time, which can be represented by the positive real numbers $\mathbb{R}_{\geq 0}$. $Time$ can also be reasoned about from the perspective of a system clock that ticks (advances time) in discrete intervals, in which case we can model it using the natural numbers $\mathbb{N}$. It is desirable to prove results about our system using continuous time as it is more general. However, some of our proved results later use a discrete time model as it allows for cleaner proofs using induction and is still general enough to cover the behaviors of systems running on a system clock.

Realistic button press behaviors will have additional constraints such as buttons cannot toggle faster than a certain frequency, and we can also make some mathematical simplifications such as making all the button press intervals left-closed [7]. With these assumptions, we can model continuous button behavior with a discrete timed model, since in each finite interval of time, given a button function, b, there are only a finite number of press and release events in b. For example, if the button behavior is $b(t) = on$ for $t \in [0, 1) \cup [2, 5)$ and $b(t) = off$ otherwise. This can be represented discretely without any loss of information as a list of pairs describing when a button gets pressed and released, e.g., $(press, 0).(release, 1).(press, 2).(release, 5)$. We can easily specify this type of list structure in Maude with its expressive typing system [7].

3.1 Button Behavior Semantics in a System

The behavior of a button we have just defined is a purely mathematical one. By itself, it has no behavior semantics. To capture the behavior of the list of button press events over time, we simply convert the list of press and release events over time into a set of delayed messages:

```
op to-msgs : PressReleaseList Oid -> Configuration .
msgs press release : Oid -> Msg .
```

The `to-msgs` operator homomorphically maps each element of the list to a message.

```
eq to-msgs(nil, O) = none .
eq to-msgs(L press(T), O) = to-msgs(L,O) delay(press(O), t(T)) .
eq to-msgs(L release(T), O) = to-msgs(L,O) delay(release(O), t(T)) .
```

The object reacting to this button press event will then receive each button-related message at the appropriate time according to the semantics of the delay operator.

4 A Pattern to Address Button Bounce Faults

With our current model of the environment (button presses as delayed messages), we are now ready to discuss how to model faults. Faults essentially add additional behavior to

the environment or system. In general, we would like to capture a fault in full generality in order to check all cases, but we also need to make enough assumptions to restrict in a realistic way the faulty behavior. Otherwise, it may become impossible to correctly design a fault-tolerant system.

4.1 Button Bounce

When a button is pressed, the button may "bounce." A button bounce is a mechanical phenomenon that occurs due to oscillations when a button is pressed. The contact voltages of the button may oscillate between high and low thresholds multiple times before stabilizing. This results in multiple erroneous button press events for only one intended button press event. Since oscillatory phenomena are usually dampened pretty quickly, there is a short time window, T^{max}_{bounce}, within which a button may bounce after it is pressed.

Of course, the basic model of button bouncing behavior can be described in the continuous time model as a relation $F_{bounce} \subseteq I_{valid} \times I_{valid}$ (implicitly parameterized by a maximum bounce time T^{max}_{bounce}) where $(b, b_f) \in F_{bounce}$ means that given an ideal input b, the faulty input b_f could result from the button bouncing fault [7]. However, with proper assumptions on the spacing of events to avoid zeno behavior, we can use F_{bounce} to define a corresponding relation on the discrete list-like representation of button press and release events. This is represented as the binary predicate `bounce-fault`. The first argument is the ideal input, and the second argument is the nonideal faulty input. The predicate returns true iff the faulty model is a possible result of button bounce faults applied to the ideal model.

```
op bounce-fault : Input Input -> Bool .
eq bounce-fault(nil,nil) = true .
```

If the last press events match, then we can remove it and look for earlier faults.

```
eq bounce-fault(I press(T), I' press(T)) = bounce-fault(I,I') .
```

If a press event occurs in the faulty model, which is later than the corresponding press event in the ideal model, then it is possibly a bounce event if it is within the T^{max}_{bounce} duration, `bounce-duration`. We can remove this event and analyze the earlier times for more faults.

```
ceq bounce-fault(I press(T), I' press(T'))
   = bounce-fault(I press(T), I')
   if T' le (T plus bounce-duration) /\ T' gt T .
```

Release events should match the ideal ones, but there might be extraneous release events generated by the bounce fault, which we can just remove and reason about the corresponding press event earlier (using the equations above). Anything that does not match the patterns described above could not have been generated by a bounce fault.

```
eq bounce-fault(I release(T),I' release(T)) = bounce-fault(I,I') .
ceq bounce-fault(I press(T), I' release(T')) = bounce-fault(I press(T),I')
if T lt T' .
eq bounce-fault(I,I') = false [owise] .
```

The current fault model is purely declarative. It is a binary relation that can be used to check whether one button input is a faulty version of another. However, this gives no means for generating a faulty model directly from a nonfaulty one. In order to have some degree of completeness in model checking analysis later, we need to have a more executable fault model; one that specifies faults as transitions and not just by a predicate. Of course, if we choose *Time* to be the real numbers, we have no hope of obtaining a set of possible faults manageable for execution purposes as there are uncountably many. However, for most practical purposes, we can obtain a fairly complete analysis just by using discrete time, mostly because systems operate based on discrete clocks anyway. Assuming a natural number model of time, a more executable fault model can be defined [7].

4.2 A Button Debouncer Pattern

Finally, we come to the most important part of our specification, namely, a formal pattern for correctly handling faulty button bounce behavior. Figure 2 shows the intuitive structure of the button debouncer. Essentially, all button inputs are filtered through a wrapper, and by properly timing button press events, we can ignore exactly the faulty bounced button press events (assuming proper spacing between normal button press events).

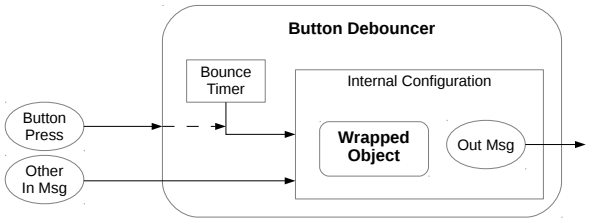


Fig. 2. The Button Debouncer Pattern

We must first describe the input theory `oth DEBOUNCED` that is required for a button debouncer. This includes the original class that the button debouncer will modify, and also parameters of the system and of the fault in order to adjust the pattern's behavioral parameters accordingly. The parameters of the theory `DEBOUNCED` can be intuitively described as follows. The class `Wrapped` is the class for the internal object that is wrapped by the button debouncer. An operator `|dest|` needs to be provided in order to know whether a message should be forwarded outside of the wrapped configuration. The constant `|t-bounce|` should be mapped to an appropriately measured constant T^{max}_{bounce}. Furthermore, another constant `t-space` is required to define the minimal time spacing between two intentional button presses. The message `press` is of course the special button press message that we want to debounce. We also add an equation in the theory

specifying that time should not be allowed to advance when a press message has not yet been handled.

```
   class |Wrapped| .
   op |dest| : Msg -> Oid .
   op |t-bounce| : -> Time .
   op |t-space| : -> Time .
   eq |t-bounce| lt |t-space| = true .
   msg |press| : Oid -> Msg .
   eq mte(|press|(O:Oid)) = zero .
```

Now, the actual pattern itself is quite straightforward. The debouncer pattern is a wrapper enclosing an object that modifies its behavior by filtering messages. Besides the internal configuration, it also adds a timer attribute, which is needed to filter the debouncing actions correctly. Note that we use parameter `|O|` as the parameter label of the theory `DEBOUNCED`.

```
(tomod DEBOUNCER{|O| :: DEBOUNCED} is
  pr RT-COMP .
  pr DELAY-MSG .

  class !Debouncer{|O|} |
     inside : NEConfiguration,
     timer : Timer .
```

The tick and mte equations are the intuitive ones, where we must tick the internal configuration according to its defined semantics as well as the timer stored in the wrapper object.

```
eq tick(< O : !Debouncer{|O|} | inside : C, timer : TM >, T)
   = < O : !Debouncer{|O|} | inside : tick(C, T), timer : tick(TM, T) > .
eq mte(< O : !Debouncer{|O|} | inside : C, timer : TM >)
   = minimum(mte(C), mte(TM)) .
```

Finally, we have the behavioral rules for the object. For receiving messages, all messages that are not a button press message are forwarded to the internal configuration. Also, all messages output from the internal object are forwarded to the external wrapper:

```
crl [forward-in] : IM < O : !Debouncer{|O|} | inside : C >
   => < O : !Debouncer{|O|} | inside : IM C >
   if |dest|(IM) == O /\ IM =/= |press|(O) .
crl [forward-out] : < O : !Debouncer{|O|} | inside : OM C >
   => < O : !Debouncer{|O|} | inside : C > OM
   if |dest|(OM) =/= O .
```

When a button press message is received, the behavior will differ based on the timer. If the timer is not set, then we have an initial button press event, which is immediately forwarded to the internal configuration. Furthermore, the timer is set for the maximum bounce duration.

```
rl [set-timer] : |press|(O) < O : !Debouncer{|O|} | timer : no-timer, inside : C >
  => < O : !Debouncer{|O|} | timer : t(|t-space|), inside : |press|(O) C > .
```

If the timer is set, then the system is within a bounce duration, and the incoming button press event is ignored.

```
crl [ignore-press] : |press|(O) < O : !Debouncer{|O|} | timer : TM, inside : C >
   => < O : !Debouncer{|O|} | inside : C >
   if TM =/= timer0 /\ TM =/= no-timer .
```

Finally, when the timer expires, the timer is removed. This is a model-specific construct that allows the time to advance.

```
crl [reset-timer] : < O : !Debouncer{|O|} | timer : TM >
   => < O : !Debouncer{|O|} | timer : no-timer >
   if TM == timer0 .
endtom)
```

4.3 Proof of Correctness of the Debouncer Pattern

The button debouncer should essentially mitigate button bounce faults, but we must make clear this notion and what it means. We essentially need to define a correspondence between ideal behavior and the debounce pattern behavior under a faulty input. We must define the two transition systems of interest and express their correspondence. First, we define appropriate projection operations. We need a message filter and a wrapper remover. π_{nf} only projects the nonfaulty messages. π_w projects the object on the inside of the wrapper. In Maude, they can be defined as follows:

```
vars C C' : NEConfiguration .
eq pi-nf(C) = pi-nonpress(C) pi-press(C, get-time(C)) .

eq pi-w(< I:Oid : PressDebouncer | inside : C >) = C .
eq pi-w(C C') = pi-w(C) pi-w(C') .
eq pi-w(C) = C [owise] .
```

Here all these operators are frozen. `pi-nonpress` projects all the components of the configuration that are not `press` messages, and `pi-press` filters all press messages that are not faulty using the defined times `T-bounce` and `T-space`, and also the timer set on the debounce wrapper to filter initial times.

Definition 1. *States of the transition system S_{ideal} are system configurations with a single instance of a wrapped object, and such that the input button press messages are spaced by at least the assumed minimal time spacing.*

States of the transition system $S_{wrapped}$ are system configurations with a single instance of a wrapped object in a wrapper object, and such that input button press messages are related to an ideal button press configuration by the button press fault F_{bounce}.

We define a relation $H \subseteq S_{ideal} \times S_{wrapped}$ by the equivalence $s_i H s_f$ iff $\pi_{nf}(\pi_w((s_f))) = s_i$ and $time(s_f) = time(s_i)$.

We now come to the theorem that shows that H defines a bisimulation between an ideal system and a faulty system with our pattern applied. Since H preserves all the states of the object, this theorem essentially states that our pattern fully masks button bounce faults for our model of input (with proper spacing between successive button presses). The full proof of the theorem can be found in [7].

Theorem 1. *The relation H is a well-founded bisimulation, and thus H defines a stuttering bisimulation between S_{ideal} and $S_{wrapped}$ when considering natural number time.*

Note that if we do not have natural number time, then it is not guaranteed that we have a bisimulation. A simple counter-example would be one where a button bounces an infinite number of times in a finite time period. Of course, this is due to Zeno behavior.

In order to remove Zeno behavior, we can make the assumption that all events are spaced at least Δt apart. This means that if we convert all times t into the natural number $\lceil t/\Delta t \rceil$, then the relation is still well founded, and the bisimulation result would still hold.

Notice that any atomic proposition AP defined on a state s_i can be lifted to a property of s_f by labelling s_f according to $\pi_{nf}(\pi_w((s_f)))$.

In addition to proving these theorems, we have also performed some model checking for simple instantiations of this pattern as an extra level of validation [7].

5 A Pattern to Address Phantom Faults

5.1 Phantom Faults

Slight disturbances in the environment (e.g. EMI, moving parts, etc.) can lead to a button being unintentionally pressed for a very short time.

The domain model is exactly the same as that for button bounce. We consider button inputs that we model as discrete messages, and an object that reacts to button inputs by consuming these messages.

A phantom button fault is a relation $F_{phantom} \subseteq I_{valid} \times I_{valid}$ (implicitly parameterized by a phantom press duration $T_{phantom}$) where faulty button presses of very short durations may occur. More precisely, $(b, b_f) \in F_{phantom}$ iff

1. $b(t) = 1 \implies b_f(t) = 1$ (an intentional button press is always registered)
2. if $b_f(t) = 1$ and $b(t) = 0$, then $t - init(b_f, t) < T_{phantom}$ (the duration of all phantom presses are bounded by $T_{phantom}$)

We can similarly construct the discrete definition of the $F_{phantom}$ relation and also the executable fault generation definitions when we are working in discrete time.

5.2 Dephantom Pattern

The pattern for handling phantom button events first requires describing the necessary parameters to fully define its behavior in the parameter theory `PHANTOMABLE`.

Like the button debouncer pattern, the dephantomizer pattern is parameterized, in this case by the `PHANTOMABLE` input theory that describes the nature of the phantom button press fault and the object which will be wrapped by the pattern. This includes a class `|Wrapped|` which specifies which object is subject to the phantom press fault. The `|dest|` operator which is again used to find which messages to forward to the outside configuration. The `|press|` and `|release|` messages which describe the actual button press events subject to phantom press faults.

```
(oth PHANTOMABLE is pr TICK-MTE-SEM .

   class |Wrapped| .
   op |dest| : Msg -> Oid .
   op |t-phantom| : -> Time .

   msg |press| : Oid -> Msg .
   msg |release| : Oid -> Msg .

   var O : Oid .
   eq mte(|press|(O)) = zero .
endoth)
```

The dephatomizer pattern takes a PHANTOMABLE theory as input and describes a wrapper pattern to mitigate phantom button press faults. The wrapper structure is very similar to the button debouncer, except for the logic of handling button presses, which is of course necessary since the fault behavior is different for the pattern.

```
(tomod DEPHANTOMIZER{|O| :: PHANTOMABLE} is
   pr RT-COMP .
   pr DELAY-MSG .

   class !PhantomIgnore{|O|} |
      inside : NEConfiguration,
      timer : Timer .

   op init-timer : -> Timer .
   eq init-timer = no-timer .

   vars T : Time .
   var O : Oid .
   var TM : Timer .
   var C : Configuration .
```

The equations below define the wrapper class and the time advancement semantics. This is exactly the same as in the button debouncer case. However, here the timer is used slightly differently to eliminate a different set of faults. The logic for the timer will be shown later.

```
   eq tick( < O : !PhantomIgnore{|O|} | inside : C, timer : TM >, T)
   = < O : !PhantomIgnore{|O|} | inside : tick(C, T), timer : tick(TM, T) > .

   eq mte( < O : !PhantomIgnore{|O|} | inside : C, timer : TM >)
   = minimum(mte(C), mte(TM)) .
```

The rule set-timer below sets the timer whenever a button press event is received. The timer is then used to make sure that the button is pressed for sufficiently long before it is actually recognized as an intentional button press event. The rule non-phantom-release decides the behavior when the system receives a release after sufficient time has elapsed, and hence the timer is disabled to no-timer. The rule phantom-release is applied when a release message is received before the timer expires. This means that insufficient time has elapsed before a button is released and it is considered a phantom event. Thus, the button press and the release events are hidden from the internal object. Furthermore, the timer is reset. The last rule reset-timer is specified when the timer expires. This means that the button press duration has just passed the threshold to be registered as a valid press. The press event is forwarded to the internal configuration.

```
   rl [set-timer] : |press|(O) < O : !PhantomIgnore{|O|} | timer : no-timer >
   => < O : !PhantomIgnore{|O|} | timer : t(|t-phantom|) > .

   rl [non-phantom-release] : |release|(O) < O : !PhantomIgnore{|O|} |
      timer : no-timer, inside : C >
   => < O : !PhantomIgnore{|O|} | inside : |release|(O) C > .

   crl [phantom-release] : |release|(O) < O : !PhantomIgnore{|O|} | timer : TM >
   => < O : !PhantomIgnore{|O|} | timer : no-timer >
   if TM =/= timer0 /\ TM =/= no-timer .
```

```
  crl [reset-timer] : < O : !PhantomIgnore{|O|} | timer : TM, inside : C >
  => < O : !PhantomIgnore{|O|} | timer : no-timer, inside : |press|(O) C >
  if TM == timer0 .
```

The last two rules for forwarding messages in and out from the internal configuration are similar to the forwarding rules for the debouncer pattern. Indeed, any wrapper that selectively filters certain messages will have forward rules of this form.

```
  var IM OM : Msg .
  crl [forward-in] : IM < O : !PhantomIgnore{|O|} | inside : C >
     => < O : !PhantomIgnore{|O|} | inside : IM C >
     if |dest|(IM) == O /\ IM =/= |press|(O) /\ IM =/= |release|(O) .
  crl [forward-out] : < O : !PhantomIgnore{|O|} | inside : OM C >
     => < O : !PhantomIgnore{|O|} | inside : C > OM
     if |dest|(OM) =/= O .
endtom)
```

5.3 Proof of Correctness of the Dephantomizer Pattern

As with the button debouncer, we would like to establish a correspondence between the execution of an ideal system and that of a system with input faults but with the pattern applied. Again, the key is to define a projection relation between the two systems. However, in this case, in addition to the projection operations, we also need to define a *time translation* on button press messages to capture the delays of the pattern.

The first transformation operation of interest is the `delay-press`, which delays all press messages by a time duration `T`. This is useful as the dephantom pattern introduces delays in processing the press messages. Because of this, a delay transformation is required to show an equivalent execution between an ideal system and a delayed system. The projection $\pi_{phantom}$ from a phantom input system with a wrapper to an ideal input system with no wrapper would be the composition `remove-small ; remove-wrapper ; delay-press`. Where `remove-small` is applied first and removes all messages whose durations are too small; `remove-wrapper` removes the pattern wrapper and exposes the internal object; and `delay-press` shifts the time of all button press events by a specific duration. Full details about each of these operator definitions can be found in [7].

Again, we use the same definitions as with the button bounce case defining the states of systems S_{ideal} and $S_{wrapped}$, but this time using the phantom fault $F_{phantom}$ to provide faulty button inputs.

Definition 2. *Define a relation $H \subseteq S_{ideal} \times S_{wrapped}$ such that $s_i H s_f$ iff $\pi_{phantom}(s_f) = s_i$ and $time(s_f) = time(s_i)$.*

We again have a bisimulation result, for which the full proof can be found in [7].

Theorem 2. *The relation H is a well-founded bisimulation, and thus H defines a stuttering bisimulation between S_{ideal} and $S_{wrapped}$ when considering natural number time.*

Notice that in this case, H still preserves all the attributes of objects but only by making the button press delivery times later in the ideal model. This means that H adds a delay into the system, which is to be expected as detecting for faulty short button presses requires the system to wait before registering the button press event.

6 A Pattern to Address Stuck Faults

6.1 Stuck Faults

When a button is pressed, it may become stuck. This may be caused by deterioration in the spring or sudden increase in friction due to deformation or adhesives. This results in a persistent logical 1 signal, even though the button was already released.

We again have another device-button interaction, and the model is entirely similar to the button bounce and phantom press cases.

A button stuck fault is a relation $F_{stuck} \subseteq I_{valid} \times I_{valid}$ such that a faulty button may be held down for longer durations than intended, or more precisely, $(b, b_f) \in F_{stuck}$ iff:

1. $b(t) = 1 \implies b_f(t) = 1$ (a button appears pressed when it is physically pressed, regardless of being stuck)
2. If $b_f(t) = 1$ and $b(t) = 0$, then there is a $t' < t$ s.t. $b(t') = 1$ and $b_f(t'') = 1$ for all $t'' \in [t', t]$ (a button can only become stuck after it has been pressed, and stays stuck for a continuous time interval).

6.2 Stuck Detection Pattern

Like the button debouncer pattern, the stuck detector pattern takes an input theory that describes the nature of the stuck button press fault. This includes a class `Wrapped` which specifies which object is subject to the stuck button press fault. The `dest` operator is again used to find which messages to forward to the outside configuration. The `press` and `release` messages describe the actual button press events subject to stuck button press faults. Furthermore, we have `t-stuck` to describe the minimal time that the button will remain stuck. The input theory for the stuck detector pattern is given as follows.

```
(oth STUCKABLE is
   pr TICK-MTE-SEM .

   class |Wrapped| .
   op |dest| : Msg -> Oid .
   op |t-stuck| : -> Time .

   msg |press| : Oid -> Msg .
   msg |release| : Oid -> Msg .

   var O : Oid .
   eq mte(|press|(O)) = zero .
endoth)
```

The stuck detector pattern is defined in the `STUCK-DETECT` module below. It takes a `STUCKABLE` theory as input and describes a wrapper pattern to detect stuck button press faults. The wrapper structure is again very similar to the button debouncer wrapper.

```
(tomod STUCK-DETECT{|O| :: STUCKABLE} is
   pr RT-COMP .
   pr DELAY-MSG .

   class !StuckDetect{|O|} |
      inside : NEConfiguration,
      timer : Timer,
      stuck-err : Bool .
```

```
op init-timer : -> Timer .
eq init-timer = no-timer .
op init-stuck-err : -> Bool .
eq init-stuck-err = false .
```

We first define the necessary attributes of the wrapper object. Besides the internal configuration, we have a timer for keeping track of when the button has been pressed passed its stuck duration. The `stuck-err` bit, when set to true represents detection of the error. The other constants define initialization values for each of the attributes.

The tick and mte rules are again similar to those for the other patterns and work by propagating the operations homomorphically to the internal configuration and timers. Their behavior on objects are defined by the equations below.

```
eq tick( < O : !StuckDetect{|O|} | inside : C, timer : TM >, T)
= < O : !StuckDetect{|O|} | inside : tick(C, T), timer : tick(TM, T) > .

eq mte( < O : !StuckDetect{|O|} | inside : C, timer : TM >)
= minimum(mte(C), mte(TM)) .
```

The rules for the behavior under button press events is just forwarding all button press and release messages normally, but setting and resetting the timers appropriately. The last rule, `stuck-event`, is applied whenever a button press event is not followed by a release within `t-stuck` time units. When this happens, the `stuck-err` is set to true to indicate detection.

```
rl [set-timer] : |press|(O) < O : !StuckDetect{|O|} | timer : no-timer, inside : C >
=> < O : !StuckDetect{|O|} | timer : t(|t-stuck|), inside : |press|(O) C > .

rl [release-event] : |release|(O) < O : !StuckDetect{|O|} | inside : C >
=> < O : !StuckDetect{|O|} | inside : |release|(O) C, timer : no-timer, stuck-err : false > .

crl [stuck-event] : < O : !StuckDetect{|O|} | timer : TM >
=> < O : !StuckDetect{|O|} | timer : no-timer, stuck-err : true >
if TM == timer0 .
```

The forward in and out rules are again similar to the previous two patterns.

```
var IM OM : Msg .
crl [forward-in] : IM < O : !StuckDetect{|O|} | inside : C >
   => < O : !StuckDetect{|O|} | inside : IM C >
   if |dest|(IM) == O /\ IM =/= |press|(O) /\ IM =/= |release|(O) .
crl [forward-out] : < O : !StuckDetect{|O|} | inside : OM C >
   => < O : !StuckDetect{|O|} | inside : C > OM
   if |dest|(OM) =/= O .
endtom)
```

6.3 Proof of Correctness of the Stuck Detection Pattern

The stuck fault is inherently lossy, so the correctness of the pattern is shown in two parts. First, if no stuck faults occur then we show that the behavior with the pattern is bisimilar to the ideal system. Second, if a stuck fault occurs, we can no longer guarantee any correspondence in behavior to the ideal case, but we can guarantee *detection* of the fault within a certain time bound.

The projection π_{stuck} from a wrapped system for stuck detection to an ideal input system with no wrapper is just simply a function `remove-wrapper`, which removes the pattern wrapper and exposes the internal object to the external configuration.

Again, we use definitions analogous to those for the button bounce case for states of S_{ideal} and $S_{wrapped}$. Although stuck faults will ruin any possibility of behavioral correspondence (since the system becomes unresponsive), we can still show that without faults our pattern does not alter the behavior of the system.

Definition 3. *Define a relation $H \subseteq S_{ideal} \times S_{wrapped}$ such that $s_i H s_f$ iff $\pi_{stuck}(s_f) = s_i$ and $time(s_f) = time(s_i)$.*

We can show that under a strict relation H that does not allow for differences in the faulty model (i.e. no stuck faults occur), then the behavior of the wrapped system in a faulty environment is bisimilar to that of the ideal system, that is, the added wrapper does not essentially change to the behavior of the system. Proof in [7].

Theorem 3. *The relation H is a well-founded bisimulation, and thus H defines a stuttering bisimulation between S_{ideal} and $S_{wrapped}$ when considering natural number time.*

However when a button does become stuck, we can no longer give any guarantees about correct behavior, but we can still detect a fault. The following theorem proves that any stuck faults will be detected by our pattern. Proof in [7].

Theorem 4. *Consider a system in $S_{wrapped}$. If we have a stuck fault such that there exist two consecutive press and release events on the input* `delay(press, t) delay(release, t')` *such that $t' - t > T_{stuck}$ then the wrapper attribute* `stuck-err` *will be set after $t + T_{stuck}$ time units.*

7 Conclusion and Future Work

The goal of this work has been to define *formal patterns*, as parameterized real-time rewrite theories, that provide provably correct guarantees of fault tolerance for commonly occuring faults in button interfaces of manually-operated devices, including medical equipment. The general technique of well-founded bisimulations [4] has been used to obtain the desired guarantees for each pattern. Since the formal specifications are executable, formal analysis by model checking has also been performed.

For future work, an important next step is to analyze the compositional behavior of multiple patterns together. Although each of the patterns have bisimulation results which is by itself composable, some of the bisimulations are conditional (such as introducing delays or adding additional fault-detection messages). In these cases the order of pattern composition can result in different system behaviors. This highly nontrivial problem of pattern composition is one of the major challenges that must be addressed before these patterns can be used for larger scale systems.

References

1. Roberto Bruni and José Meseguer. Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.*, 360(1):386–414, 2006.
2. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude – A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
3. F. Durán and J. Meseguer. The Maude specification of Full Maude. Technical report, SRI International, 1999.
4. J. Meseguer, M. Palomino, and N. Martí-Oliet. Algebraic simulations. *J. Log. Algebr. Program.*, 79(2):103–143, 2010.
5. José Meseguer. Membership algebra as a logical framework for equational specification. In *WADT '97: Selected papers from the 12th International Workshop on Recent Trends in Algebraic Development Techniques*, pages 18–61, London, UK, 1997. Springer-Verlag.
6. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.
7. Mu Sun. Formal patterns for medical device safety. *Doctoral Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign https://dl.dropboxusercontent.com/u/54321762/mu-thesis.pdf*, 2013.

A Formal Heartbeat Pattern for Open-Loop Safety of Networked Medical Devices

Mu Sun José Meseguer Lui Sha musun(at)illinois.edu
meseguer(at)illinois.edu lrs(at)illinois.edu

University of Illinois at Urbana-Champaign
Illinois, USA

Abstract. Networking of computers and devices have improved parallelism, fault tolerance, and the overall capabilities of real-time systems. However, the power of networking does not come for free as we are left to deal with new faults of message loss, message delay, and message corruption. Generally, in order to abstract away faults in networked systems, we either assume that the system is not time sensitive and information can be retransmitted indefinitely, or we must have dedicated reliable networking hardware and appropriate protocols to remove the faults entirely. In the paper, we present a formally specified heart-beat pattern for use in real-time safety-critical systems with unreliable networks. We use parameterize modules to specify our pattern generically, not tied to the particular system being designed.

1 Introduction

Networking has always been a key element in improving the scalability of cyber-physical systems. When used correctly, networking can improve scalability and performance by increasing parallelism, improve the fault tolerance of systems by introducing redundancy, and improve the overall computing power by linking disjoint computing elements together. However, none of these benefits of networking comes for free, and naive usage of network technology can easily have the opposite effect of descreasing performance and fault-tolerance.

Networking introduces many new problems that we must address before having a correctly functioning system:

- Message delivery is not reliable, since messages can be lost on the network due to noise or other issues.
- Message delays are more variable, since messages may be routed in many ways.
- Failure dependencies increase, since the failure of one network component can causally disrupt all other components across the network.

In safety-critical systems such as avionics, an ultra reliable real-time network is used to ensure timely communication between components. However, in medical systems, these hard real-time network guarantees are much harder to achieve

since medical devices need to be mobile (across hospital rooms) and setting up cables to connect different medical devices can become very error prone and hazardous. Thus, modern medical system designs prefer to use wireless connections to set up devices and to improve mobility. However, this inherently reduces the network reliablity of medical systems compared to other systems. This poses a safety challenge that must be addressed in medical system designs. We call this the problem of *open-loop safety*, which means maintaining safety in a system even in the situation when devices become disconnected from each other due to network failures.

In this paper, we describe a *heartbeat formal pattern* for networked medical systems and more generally CPS systems operating over unreliable networks. The heartbeat pattern creates a fault-tolerance layer that can still ensure some real-time safety interlocks when messages are dropped. We use the term *ideal system* to describe a system design that is correct without considering message loss faults. Our heartbeat pattern itself is then a *theory transformation* from the ideal system to a system that operates in an environment with message loss faults. Our pattern should have the following properties:

- *Generic*: our pattern applies to any system that can be modelled as a real-time object-oriented actor model with a sender-receiver pair.
- *Formally defined preconditions*: the conditions required for using our pattern correctly are captured in the form of an input theory.
- *Proven to be correct*: when preconditions are met and no faults occur, we formally prove that our pattern preserves the behavior of the ideal system.
- *Proven to be robust*: when preconditions are met and faults occur, we formally prove that our pattern guarantees delivery of critical messages, and thus, can be used to guarantee a certain level of safety.

2 Safe Networked Device Communication

Network failures must be taken into account in almost all distributed systems. General solutions usually involve time stamping, acks, and resending. However, in a real-time system, waiting for resends is a luxury that in general cannot be afforded. A real-time control system must be able to remain safe, which includes making timely decisions, even if the entire network permanently fails and all devices become disconnected. This is especially problematic when devices are tightly coupled and must coordinate actions in a timely manner. For the patterns in this section, we are solely focused on the following problem:
Context: A set of communicating devices whose behaviors need to be coupled in certain ways.
Problem: Critical messages for device coordination could be lost due to a faulty network.
Domain Model: We use timed object-oriented actor models in Maude to model our system. A configuration is a multiset of messages and objects. The rewrite (state transition) rule for potential message loss of a *LossyMsg* in a configuration is just:

```
M:LossyMsg C:Configuration => C:Configuration
```

2.1 Example Instances

1. During laser-airway surgery, the oxygen on a mechanical ventilator must be turned-off before using the laser. If the laser is on with oxygen flowing, then a dangerous airway fire can result. Furthermore, if the oxygen does not turn back on within a certain time, then hypoxia can result for the patient. The laser and oxygen devices must coordinate their behaviors closely over the network to prevent harm to the patient.
2. A patient is being administered calcium chloride after a heart attack through an infusion device. Later the patient develops blood acidosis and needs to be given sodium bicarbonate. The calcium chloride infusion must be stopped before sodium bicarbonate can be started; otherwise calcium bicarbonate precipitate can form. Again, these two infusion devices need to communicate to prevent harm to the patient.

2.2 Heartbeat Protocol for Fail-Safe Communication

The heartbeat protocol focuses on single communication channels between two nodes. We assume that a device behavior is being controlled/coordinated by some message sender (either a supervisory device or another medical device). In order for the heartbeat protocol to make sense, we make the following assumptions on the receiving device D:

1. D has a set of well-defined global safe states S_{safe}. In these states it is assumed that there will not be any adverse interactions between D and other device states.
2. There is a critical message M_{crit} such that after receiving M_{crit}, D will transition to a state in S_{safe}.

Given that M_{crit} can be lost, the heartbeat pattern makes sure that D still enters a safe state by constructing a new message $re(M_{crit})$. This new message is resent periodically as a heartbeat message when M_{crit} has not yet been sent. Both the original sender and receiver are encapsulated into meta-objects. The sender meta-object, will periodically retransmit $re(M_{crit})$ until M_{crit} is sent by the inner sender. The receiver meta-object will forward a M_{crit} message to the inner receiver, either when M_{crit} is received or when $re(M_{crit})$ has not been received for a certain period of time:

```
rl < sender : SendWrapper | inner : C M-crit >
   => < sender : SendWrapper | inner : C, resend : no-timer > M-crit .
rl < sender : SendWrapper | inner : C, resend : t(0) >
   => < sender : SendWrapper | inner : C, resend : t(resend-time) >
   re(M-crit) .
rl M-crit < receiver : RecvWrapper | inner : C >
   => < receiver : RecvWrapper | inner : M-crit C, timeout : t(timeout-time) > .
rl < receiver : RecvWrapper | inner : C, timeout : t(0) >
   => < receiver : RecvWrapper | inner : M-crit C, timeout : t(timeout-time) > .
```

Correctness: For this pattern, it can be shown that the receiver will always receive a message M_{crit} (at a delay of at most $T_{timeout}$). Thus, even during network disconnects the receiver device D will transition to a state in S_{safe}. Of course, the device D can also preemptively transition to a state in S_{safe} due to message loss; however, this is still safe. It can also be shown true that if there is no message loss, there is a bisimulation between the device behavior using the heartbeat protocol and the device behavior behavior without the heartbeat protocol.

3 Conclusion and Future Work

Our heartbeat pattern addresses an important safety problem for medical devices connected by an unreliable network, which can be described as an open-loop safety problem. Our pattern is generic in that in can be applied to any sender and receiver pair that satisfies the preconditions stated in the pattern's parameter theory. It as also provably correct under nonfailure conditions and provably safe under message loss failures.

In this work, our pattern still makes some ideal assumptions that should sbe addressed in future work, including the addition of clock skews and variable message delays. Furthermore, the generalization of our pattern to multiple sender and receiver configurations can also be considered. On the modeling perspective, our model specification are still limited to object-oriented actor systems. It will be interesting to see how to define the same pattern for more hardware-oriented models of communication.

References

A Formal Semantics of the OSEK/VDX Standard in $\mathbb{K}$ Framework and its Applications*

Min Zhang[1], Yunja Choi[2], Kazuhiro Ogata[1]

[1] Research Center for Software Verification,
Japan Advanced Institute of Science and Technology (JAIST)
{zhangmin,ogata}@jaist.ac.jp
[2] School of Computer Science and Engineering, Kyungpook National University
yuchoi76@knu.ac.kr

Abstract. The OSEK/VDX is an international standard of automobile operating systems, which are typical safety-critical systems that require extensive safety analysis and verification. Our previous work has shown formal methods are useful to verify the safety of both the OSEK/VDX-based operating systems and applications. Using formal methods requires formal semantics of the OSEK/VDX standard. In this paper, we present a formal semantics of the standard defined in $\mathbb{K}$, a rewrite-based formal semantics framework. With the formal semantics, we can (1) verify user-defined applications by symbolic execution, and (2) automatically generate test cases for testing of the OSEK/VDX-based operating systems. Compared with existing formal semantics of the standard, the formal semantics defined in $\mathbb{K}$ is more flexible and generic. This work also shows that $\mathbb{K}$ is not only used for formalizing the semantics of programming languages, but also for automobile operating systems.

1 Introduction

The OSEK/VDX is an international standard of developing automobile operating systems [1]. An automobile operating system is a piece of safety-critical software to manage resources and applications which run on the system to control electrical devices in automobiles. Its safety should be extensively analyzed and verified. To implement an OSEK/VDX-based operating system, the traditional approach is to develop both the kernel and applications following the standard, and compile them together to generate an executable system. The system must be tested extensively for safety [2]. This approach is effort-consuming and prone to errors in that modification to source code usually requires recompilation and testing requires complete suite of test cases, which usually are difficult to build. Our previous work shows that using formal methods is an effective approach

* This research was supported by Kakenhi 23220002, Japan, and by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (NIPA-2013-H0301-13-5004) supervised by the NIPA(National IT Industry Promotion Agency).

to both safety verification of OSEK/VDX-based applications [3] and test case generation [4, 5], complementary to the traditional testing-based approach.

Using formal methods requires formal semantics of the OSEK/VDX standard. In this paper, we present an executable formal semantics of the standard, which is defined in $\mathbb{K}$, a rewrite-based formal framework [6]. We choose $\mathbb{K}$ for its flexibility, simplicity and tool-support. $\mathbb{K}$ allows user-defined data types and supports formalization of infinite-state systems. Especially, $\mathbb{K}$ provides tool-support to automatically generate interpreter and state-space explorer based on the defined semantics. Another advantage of using $\mathbb{K}$ is that it does not require extra effort to transform user-defined applications into corresponding formal definition in $\mathbb{K}$ in order to use the formal semantics. In this sense, the formal semantics of the standard in $\mathbb{K}$ is more flexible and generic than those formalized in Promela [7] and NuSMV [5], which have restriction on the number of tasks, resources, events in OSEK/VDX-based operating systems, and also need extra effort to instantiate the semantics with user-defined applications.

The benefit from this formal semantics is multifold. Firstly, it can be used to verify user-defined applications by integrating the formal semantics with the semantics of the language in which the applications are implemented. Secondly, it can be used to automatically generate test cases for the testing of OSEK/VDX-based operating systems. This work also shows that $\mathbb{K}$ is not only used for formalization of the semantics of programming languages, but also for the formalization of automobile operating systems. To the best of our knowledge, this is the first work to formalize operating systems in the $\mathbb{K}$ framework.

Organization of the paper: Section 2 introduces the background and our overall approach. Sections 3 and 4 describe the OSEK/VDX standard and $\mathbb{K}$. Section 5 shows the formalization of OSEK/VDX in $\mathbb{K}$. Section 6 demonstrates two applications of the formal semantics, i.e., symbolic execution and test case generation. Sections 7 and 8 mention some related work and conclude the paper.

2 Background and Overall Approach

The OSEK/VDX standard is a generic description which is mandatory for any implementation of an OSEK/VDX operating system. It concerns the general

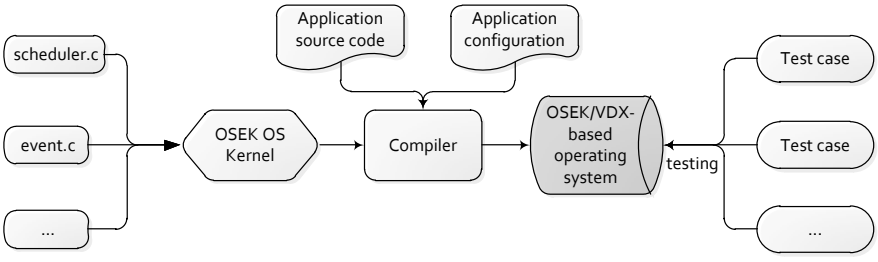


Fig. 1. Traditional approach to implementing an OSEK/VDX-based operating system

description of the strategy and functionality, standardized application programming interface (API), resource management, event mechanism, etc. Fig. 1 depicts the traditional process of implementing OSEK/VDX-based operating systems [8]. An OSEK/VDX-based operating system is built out of a kernel which includes basic functionalities described in the standard such as scheduler, APIs, etc., and a group of applications which interact with the kernel through APIs. An application includes a configuration of resources, tasks, and events that are defined in OIL (OSEK Implementation Language) and source code for each task of the application. They are compiled together and an executable operating system is developed. The system is then extensively tested for safety. There are two major problems with the traditional approach. One is that the whole system must be re-compiled and re-tested due to every change to either the kernel or applications. It is costly in terms of both effort and time. Another problem is that it is prone to errors due to the ambiguity of the standard which in written in natural language and the lack of test cases. A suite of comprehensive test cases are necessary to detect potential errors in a system, however it is not an easy task to build such a suite of test cases.

Using formal methods is an effective means of developing reliable OSEK/VDX-based operating systems, complementary to the traditional one. Fig. 2 shows the overview of our formal approach to the verification of OSEK/VDX-based applications and test case generation for the testing of OSEK/VDX-based operating systems. The OSEK/VDX standard, including features such as task scheduling, resource management, or event mechanism is formalized. To verify user-defined applications, the semantics of the programming language in which tasks are implemented should also be formalized. User-defined applications can be symbolically executed with the integration of the two formal semantics. By symbolic execution, we can verify the execution result and detect potential errors such as deadlock in applications. For test case generation, users only need to provide a configuration of tasks, resources and events, and the constraints that generated test cases should satisfy, such as the number of APIs for each task. Test cases are

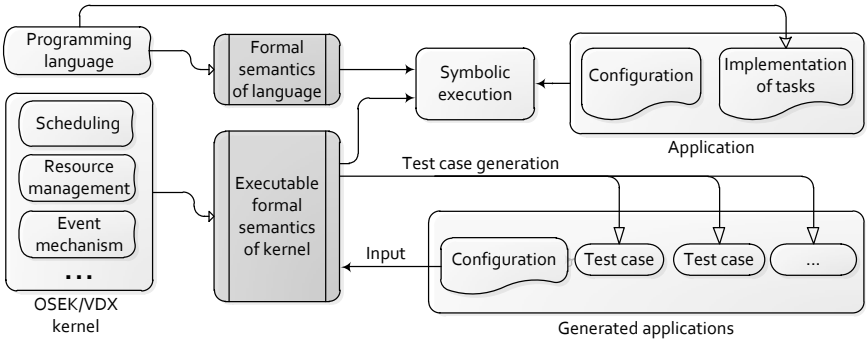


Fig. 2. The framework of our formal approach

automatically generated based on the formal semantics of the kernel. Generated test cases can be used for OSEK/VDX-based operating systems by checking whether the result obtained by running each test case on practical system is the same as the expected result.

3 The OSEK/VDX Standard and OIL

As mentioned earlier, the OSEK/VDX standard generically describes all mandatory requirements of an OSEK/VDX-based operating system. In our current formalization we only consider some fundamental parts in the standard such as task scheduling, resource management, event mechanism, error handling, etc., and leave others as future work.

Task and task scheduling Task is the basic building block of an OSEK/VDX application. Multitask is one of the basic requirements of OSEK/VDX-based operating systems. The OSEK/VDX standard specifies two kinds of tasks, i.e., *basic task* and *extended task*. Fig. 3 shows the state transitions of basic tasks and extended tasks. A basic task has three states, i.e., *ready*, *running* and *suspended*, while an extended task has a *waiting* state besides the three. The difference between them is that the extended tasks can wait for events during execution by using the system call *WaitEvent*, while the basic tasks cannot. Calling *WaitEvent* may result in a *waiting* state, and the release of the processor. The processor can be reassigned to a lower-priority task without the need to terminate the running extended task.

Tasks are controlled by the scheduler. The scheduler decides on the basis of the task priority which is the next of the *ready* tasks to be transferred into the *running* state. The OSEK/VDX standard provides two scheduling policies, i.e., *full preemptive* and *non preemptive* scheduling. By full preemptive scheduling, a running task may be rescheduled at any instruction by the occurrence of trigger conditions pre-set by the operating system, such as successful termination of a task, and activating a task. The running task is put into the ready state, as soon as a higher priority task gets ready.

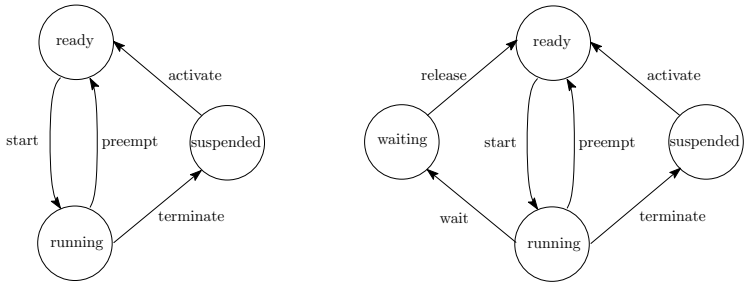


Fig. 3. The state model of basic task (left) and extended task (right)

Resource management and priority ceiling protocol Resource management is used to co-ordinate concurrent accesses of several tasks with different priorities to shared resources. It ensures that two tasks can never occupy the same resource at the same time, deadlocks will never occur by use of these resources, and access to resources never results in a *waiting* state.

There are some restrictions when using resources. When occupying a resource, the task should not call some APIs such as *TerminateTask*, which may cause rescheduling after they are called. If a task is occupying multiple resources, these resources must be released in LIFO order.

However, under these restrictions it is possible that a lower-priority task may delay the execution of higher-priority task, which is called *priority inversion*. An example about it can be found in [1]. To avoid priority inversion, OSEK prescribes the OSEK Priority Ceiling Protocol (PCP). The protocol requires that each resource has a ceiling priority which is statically assigned at the system generation. Basically, the priority shall be set at least to the highest priority of all tasks that access that resource. If a task requires a resource, and its current priority is lower than the ceiling priority of the resource, the priority of the task is raised to the ceiling priority of the resource. If the task releases a resource, the priority of the task is reset to the one before requiring that resource.

Event mechanism Tasks in OSEK/VDX-based operating systems are synchronized by events. Events are the criteria for the transition of extended tasks from the *waiting* state to the *ready* state (see Fig. 3). Events are not independent objects, but assigned to extended tasks. An event can be assigned to multiple extended tasks, and each extended task has a definite number of events.

When activating an extended task, its events are cleared by the system. All tasks can set any event of any non-suspended extended task, but only the owner can clear its events. Details about event mechanism can be referred to [1].

OSEK Implementation Language (OIL) OIL is used to configure tasks, resources, events and their relations in an OSEK/VDX-based operating system [8]. Fig. 3 shows an example of how to declare resources, events and tasks in OIL. It says that in the corresponding application there is a resource named `r1`, an event named `e1`, and a task named `t1`. Resource `r1` is declared as a standard

```
RESOURCE r1 {
  RESOURCEPROPERTY = STANDARD;
};

EVENT e1 {
  MASK = AUTO;
};
```

```
TASK t1{
  AUTOSTART = true;
  PRIORITY = 3;
  SCHEDULE = FULL;
  RESOURCE = r1;
  EVENT = e1;
};
```

Fig. 4. An example of configuration in OIL

resource. Event `e1` is declared with a mask as `AUTO`. Event mask is an integer number. If a mask is set `AUTO`, one bit is assigned to it. The statements in the configuration of task `t1` say that the task should be automatically started (put into *ready* state) after the initialization of operation system. The priority of the task is 3, and it is preemptable (indicated by `FULL`). The last two statements in `t1` mean that task `t1` can access resource `r1`, and it has the event `e1`.

4 The $\mathbb{K}$ Framework

$\mathbb{K}$ is a rewrite-based executable semantics framework, in which programming languages, calculi, as well as type systems or formal analysis tools can be defined [6]. $\mathbb{K}$ definition of a semantics is automatically translated into Maude rewrite theories [9] for execution and formal analysis purposes. The $\mathbb{K}$ tool has been used to formalize some practical programming languages such as C [10]. Some analysis tools have also been defined in $\mathbb{K}$ such as a type checker and type inferencers [6].

Semantics in $\mathbb{K}$ is defined using labeled and potentially nested cell structures and $\mathbb{K}$ (rewrite) rules. The cell structure is called a *configuration*, which is used to represent system or program state. In this paper, we call it $\mathbb{K}$ configuration to differ from the configuration of OSEK/VDX-based applications. There are two types of $\mathbb{K}$ rules: *computational rules*, which count as computational steps, and *structural rules*, which do not count as computational steps. The role of structural rules is to rearrange the configuration so that computational rules can match and apply. They correspond to the equations and rewrite rules respectively in rewriting logic [11].

The formal definition of a programming language in $\mathbb{K}$ automatically yields an interpreter for the language, and program analysis tools such as a state-space explorer, with which we can verify the result of a program in that language by symbolically executing it with the interpreter, and explore all possible results (under the condition that the state space is finite and reasonably small) by searching or model checking.

5 Formalizing the OSEK/VDX Standard in $\mathbb{K}$

In this section, we explain our approach to formalizing the OSEK/VDX standard in the $\mathbb{K}$ framework [3].

5.1 $\mathbb{K}$ configuration of the OSEK/VDX

The $\mathbb{K}$ configuration of a running OSEK/VDX-based operating system consists of over 40 nested cells. Fig. 5 shows part of them. Each cell has a label. The label ended with $*$ indicates that there can be multiple such cells. A cell that does not have nested cells is a unit cell, storing a term which represents a piece of information of a state. In the brackets is the type of the terms in Fig. 5.

[3] Some details are omitted due to the limitation of space. The complete formalization, $\mathbb{K}$ source code and the examples mentioned in Section 6 are available at the webpage `http://www.jaist.ac.jp/~zhangmin/osek-formal.html`.

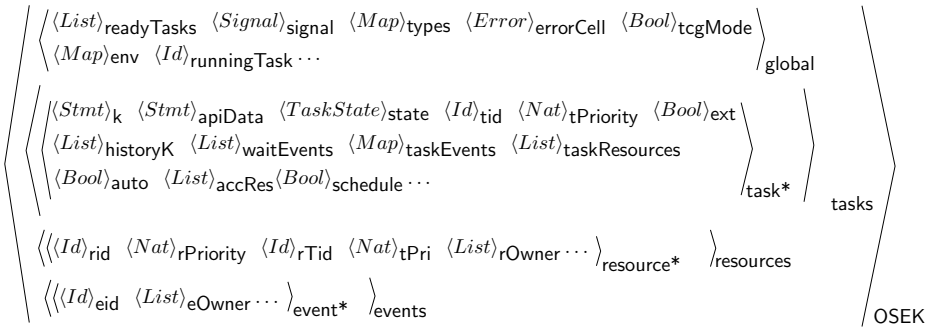


Fig. 5. $\mathbb{K}$ configuration of the OSEK/VDX standard

We take the task cell for example. There is both static and dynamic information of a task represented by the nested cells in task. Static information is that configured by users such as task ID in cell tid, priority in cell tPriority, its source code in cell apiData, whether the task is an extended one in cell ext, etc. Dynamic information is that which changes during the execution of operating system such as the next statement to be executed in cell k, the list of events which the task is waiting for in cell waitEvents, the events owned by the task and their status (*set* or *clear*) in cell taskEvents, etc. We do not explain all of the cells due to the space limitation. Some will be explained later when needed.

5.2 Formalization of the scheduler

In our formalization, we only consider *full preemptive* scheduling. As mentioned earlier, the occurrence of trigger conditions such as termination of a task will cause operating system to reschedule tasks. We define a type `Signal` and a constant `schedule` of it. We use a cell with label signal to store the occurrence of such trigger conditions. When there is a signal `schedule` in the signal cell, it indicates that some trigger condition has just occurred. Operating system must first handle it before executing any task. We define a set of $\mathbb{K}$ rules to specify the scheduler. The main one is as follows:

```
rule <signal> schedule => .</signal> <runningTask> I' => I </runningTask>
     <readyTasks> (< I,N >, L) => add2Head(I',N',L) </readyTasks>
     <task> <schedule> FULL </schedule> <state> running => ready </state>
            <tid> I' </tid> <tPriority> N' </tPriority> ... </task>
     <task> <tid> I </tid> <state> ready => running </state> ... </task>
when N >Int N' [transition]
```

The rule specifies how a $\mathbb{K}$ configuration changes before and after scheduling. Before scheduling, there is a signal `schedule` in cell signal. In cell readyTasks there is the list of ready tasks in a descend order by their priority. If there are two or more ready tasks with the same priority, they are ordered by the time

when they get ready. However, the task which is preempted from running is considered as the oldest one among the ready tasks of the same priority [1]. In the cell readyTasks, `< I,N >, L` represents that task `I` is the oldest one with the highest priority `N` among the ready tasks. The cells nested in the first task represent that task `I'` is the present running task with priority `N'` and it is preemptable, indicated by the value `FULL` in the cell schedule. If task `I` has a higher priority than task `I'`, i.e., `N >Int N'`, `I'` is preempted, and `I` becomes running. After being preempted, `I'` is changed into ready state. It is added to the head of the sub-list of the ready tasks which have the same priority as `I` in the list of ready tasks by function `add2Head`.

5.3 Formalization of resource management

As depicted in Fig. 5, each resource is represented as a cell with label resource, which consists of unit cells for the resource identifier, ceiling priority, and a list of tasks that can access the resource. It also contains two unit cells which are dynamically created when the resource is allocated to a task. The two unit cells are used to store the identifier of the task to which the resource is allocated, and the priority of the task before it gets the resource.

Tasks access resources by two APIs, i.e., *GetResource* and *ReleaseResource*. As mentioned earlier, there are restrictions when accessing resources. Such restrictions together with the Priority Ceiling Protocol should be reflected by the formal semantics of the two APIs. For instance, the main rule defined for *GetResource* is as follows:

```
rule <signal> schedule => .</signal> <runningTask> I' => I </runningTask>
     <readyTasks> (< I,N >, L) => add2Head(I',N',L) </readyTasks>
     <task> <schedule> FULL </schedule> <state> running => ready </state>
            <tid> I' </tid> <tPriority> N' </tPriority> ... </task>
     <task> <tid> I </tid> <state> ready => running </state> ... </task>
when N >Int N' [transition]
```

In the cell k, there is a list of APIs to be executed by task `I`. The API to be executed next in the list is `GetResource(R)`, where `R` is a resource ID. The cell `tPriority` stores the present priority of task `I`, and the cell `accRes` stores the list `L` of resources which task `I` is accessing. Resource `R` has a ceiling priority `N2`. `L'` in the cell rOwner represents the list of tasks that can access resource `R`. The condition (following the keyword `when`) is true when task `I` can access but is not accessing resource `R`. When the condition is true, `R` is allocated to `I`. According to the Priority Ceiling Protocol, if the present priority of task `I` is lower than the resource's ceiling priority, it is raised to the ceiling priority, as defined in the cell tPriority. `R` is added to the head of the list of resources being accessed by `I`. In the cell resource for `R`, two cells rTid and tPri are created, storing the task's ID and present priority, i.e., `I` and `N'`, respectively. The present priority should be stored because when task `I` releases the resource by *ReleaseResource*, its priority should be reset to the one before it gets the resource. The semantics of *ReleaseResource* can be defined likewise. We omit the details of it in the paper.

5.4 Formalization of event mechanism

Each event is represented by an event cell as shown in Fig. 5. An event cell only consists of two sub-cells, storing the static information of the event, i.e., the event name in cell labeled by eid, and a list of owners (tasks) to which the event can be accessed in cell labeled by eOwner. Because event is not an independent object, but is assigned to extended tasks. We declare a new cell with label taskEvents for each extended task, storing the status of events that are assigned to the task. The status of an event is either *set* or *clear*, indicating that the event is set or cleared respectively. We declare a type `EventStatus` and two constants `SET` and `CLEAR` of it to represent the two corresponding status.

There are four APIs associated to events, i.e., *GetEvent*, *SetEvent*, *WaitEvent* and *ClearEvent*, with which tasks can get, set, wait and clear specific events. We take *SetEvent* for example to show how to define its semantics in $\mathbb{K}$. *SetEvent* takes a task ID and an event name. The state of the task specified in the API is transferred to *ready* state, if the event specified in the API is one of the events which the task was waiting for. This can be formalized by the following $\mathbb{K}$ rule:

```
rule <task> <state> running </state> <k> SetEvent(I,E); => . ...</k>
  ...</task> <task> <tid> I </tid> <state> waiting => ready </state>
  <waitEvents> L => . </waitEvents> <tPriority> N </tPriority>
  <taskEvents>... (E |-> (CLEAR => SET)) ...</taskEvents> ...</task>
  <readyTasks> L' => add2Tail(I,N,L') </readyTasks>
  <signal> . => schedule </signal>  when (E in L)  [transition]
```

The rule says that `SetEvent(I,E);` is the next API to be executed by a running task, where `I` is a task ID and `E` is an event name. Task `I` is in the *waiting* state, and is waiting for a list `L` of events. If `E` is in the list, task is transferred to *ready* state, and it does not wait for any events. Thus, the list `L` is changed into an empty one, represented by `L => .` in the cell waitEvents. The status of event `E` is changed into `SET` in the cell taskEvents. Task I is added to the tail of the sub-list of ready tasks which have the same priority as `I` in the list of ready tasks by function `add2Tail`. At the same time, the `schedule` signal is fired to invoke the scheduler.

If task `I` is not waiting for the event `E`, the event is simply set after the API is called. We omit the formal definition of it and those of other three event-related APIs in the paper.

5.5 Formalization of error handling

There are pre-defined errors in the OSEK/VDX standard. Such errors should be handled correctly when an operating system is implemented. For instance, an error will occur when a task tries to terminate itself while occupying some resources, which is strictly forbidden. If an error is raised, a specific error code should be returned. However, the standard does not specify how to handle such errors, and it is up to system developers.

We formalize errors by a specific function called `Error`, which takes four arguments, i.e., an error code, the identifier of the task which causes the error, the API that causes the error, and a string to provide detailed information of the error. When an error occurs, an error cell errorCell as shown in Fig. 5 is created with a term constructed by the `Error` function in it. At the same time, an error signal is inspired and put in the signal cell. The following rule shows an example of the formalization of the error which is caused when a task tries to terminate while still occupying resources.

```
rule <task> <state> running </state> <k> TerminateTask(); </k>
     <tid> I </tid> <accRes> ListItem(R) ... </accRes> ... </task>
     <signal> . => stop </signal>
     (.Bag => <errorCell> Error(E_OS_RESOURCE, I, TerminateTask();,
     "Task cannot terminate when occupying resources!") </errorCell>)
[transition]
```

The term in cell accRes is the list of resources that are currently occupied by the task. In the above rule the list is not empty when the API `TerminateTask();` is to be executed in the next step. In the cell errorCell, `E_OS_RESOURCE` is an error code which is pre-defined in the standard.

5.6 Formalization of OIL

We formalize OIL in $\mathbb{K}$ in order to support user-defined configurations. $\mathbb{K}$ is well suited to formalize programming languages. OIL can also be naturally formalized in $\mathbb{K}$ like other languages such as C. One difference is that the semantics of OIL is formalized as structural rules, instead of computational rules. That is because OIL is a configuration language, which is used for configuring resources, tasks, events, etc. in a system, but not for computation or execution.

Given an OIL program, a $\mathbb{K}$ configuration is instantiated based on the declarations of resources, tasks, and events in the program. For instance, for each resource which is declared as shown in Fig. 3, a resource cell is created, which consists of four unit cells for resource identifier, its ceiling priority (0 at initial), the resource property, and the list of tasks that can access it (empty at initial). The following rule specifies the creation of a resource cell according to a declaration of a resource. The condition means that `I` is not used as an identifier for other tasks, events and resources.

```
rule <k> (RESOURCE I:Id { RESOURCEPROPERTY = RP; } ; => .) ... </k>
    <resources>(. => <resource> <rid> I </rid> <rPriority> 0 </rPriority>
       <rProperty> RP </rProperty> <rOwner> .List </rOwner> </resource>)
       ... </resources> <types> M:Map => (I |-> resource) M </types>
    <signal> . </signal> when notBool $hasMapping(M,I) [structural]
```

The ceiling priority of the resource and the list of tasks are calculated when tasks are initialized. If a task is declared to own a resource as shown in Fig. 3, it is added to the list. If the current ceiling priority (initially 0) is less than

the priority of the task, it is raised to the priority of the task. Corresponding structural rules can be defined likewise. We omit the details in the paper.

6 Applications

In this section, we describe two applications of the formal semantics of the OSEK/VDX standard, i.e., to verify OSEK/VDX applications by symbolic execution, and to generate test cases by searching.

6.1 Verification of OSEK/VDX applications by symbolic execution

There may be multiple tasks running in an application. It is necessary to verify that tasks can correctly synchronize and deadlock can never happen. Suppose that there are only two tasks t_1, t_2 in an application and two events e_1, e_2.Task t_1 is waiting for e_1 in order to set event e_2, while t_2 is waiting for e_2 in order to set event e_1. Both the two tasks are in *waiting* state, leading to deadlock.

An OSEK/VDX application consists of two parts: one is application configuration describing the basic information of resources, tasks, events, etc, in the application defined in OIL, and source code of each task in some programming language such as C. Thus, it requires to formalize the semantics of a specific programming language in which tasks are implemented, as shown in Fig. 2.

To demonstrate the feasibility of verifying OSEK/VDX-based applications, we use a simple C-like imperative programming language whose semantics has been formally defined in $\mathbb{K}$ [12]. We integrate the semantics of the language and the semantics of the OSEK/VDX standard. With the integrated semantics, we verify the OSEK/VDX applications that are implemented in the simplified language.

Fig. 6 shows a simplified application which is used to monitor tire pressure [3]. There are four tasks with different priorities. Task `MT` is used to repeatedly activate task `RT` (line 34), which collects data from tire sensor and then activates task `ST`. Task `ST` puts the collected data into buffer (line 47) and activate task `PT` to process the data (line 49). The synchronization between task `RT` and `ST` is achieved by an event `evt`. Task `ST` has to wait for the event until the event is set by `RT` (line 45, 41).

The application is supposed to run repeatedly. However, we found a deadlock occurred by symbolically executing the application with the integrated semantics. The returned result shows that the error occurred because the state task `RT` tries to activate task `ST`, while task `ST` is in *waiting* state [4]. The execution path shows that after `RT` activates `ST` (line 39), `ST` starts to run because it has a higher priority than `RT`. However, `ST` goes to *waiting* state because `evt` is not set (line 45). The scheduler selects `RT` to run because among the ready tasks, i.e., `RT` and `MT`, `RT`'s priority is the highest. However `RT` does not set the event because `evt`

[4] In OSEK/VDX standard, there is a maximum number of task activation. In our experiment, we assume the maximal number is 1. In this case, an error occurs because it violates the maximum activation count.

```
1 EVENT evt {
2    MASK = AUTO;
3 };
4 RESOURCE BUFF{
5   RESOURCEPROPERTY = STANDARD;
6 };
7 TASK MT {
8   PRIORITY = 4;
9   AUTOSTART = true;
10  SCHEDULE = FULL;
11 };
12 TASK RT{
13  PRIORITY = 6;
14  AUTOSTART = false;
15  SCHEDULE = FULL;
16 };
17 TASK ST{
18  PRIORITY = 8;
19  AUTOSTART = false;
20  SCHEDULE = FULL;
21  EVENT = evt;
22  RESOURCE = BUFF;
23 };
24 TASK PT{
25  PRIORITY = 10;
26  AUTOSTART = false;
27  SCHEDULE = FULL;
28  RESOURCE = BUFF;
29 };
```

a. Configuration

```
30 int data;
31 int buff;
32
33 TASK MT{
34   while(true){ActivateTask(RT);}
35 };
36 //Terminate
37 TASK RT{
38   //get data from tire sensor
39   ActivateTask(ST);
40   if(data!=0)
41     {SetEvent(ST,evt);  }
42   TerminateTask();
43 };
44 TASK ST{
45   WaitEvent(evt);
46   GetResource(BUFF);
47   buff=data;
48   ReleaseResource(BUFF);
49   ChainTask(PT);
50 };
51 TASK PT{
52   //Read data from buff
53   GetResource(BUFF);
54   // process data
55   buff=0;
56   ReleaseResource(BUFF);
57   TerminateTask();
58 };
```

b. Implementation of tasks

Fig. 6. The configuration and source code of an OSEK/VDX application

`data` is 0 (line 40). It just terminates itself (line 42). `MT` is the only ready task, which is selected to run by the scheduler. It activates `RT` (line 34), and `RT` tries to activate `ST` (line 39). However, `ST` is in the *waiting* state, leading to the deadlock. The problem is caused by the code at line 40 and 41 because `evt` cannot be always set after `ST` is activated. We can fix it by moving `ActivateTask(ST);` to the block of `if` condition. We execute the revised application by *searching*. No solution is found, which means that there is no deadlock state from which the system cannot proceed further.

OSEK/VDX-based applications are usually implemented in C. The semantics of C has been formalized in $\mathbb{K}$ [10]. We believe that by integrating the semantics of C and the standard we can verify more complicated OSEK/VDX-based applications implemented in C, which is one piece of our future work.

6.2 Using the formal semantics for test case generation

Testing is still the main approach to conformance checking of OSEK/VDX-based operating systems. However, it is practically impossible to test all possible combinations of APIs and one solution is to analyze the constraints among APIs and to generate automatically test cases that satisfy these constraints [7, 5]. Given some constraints and a configuration of tasks, resources, and events, the

test case generation problem is to generate a sequence of APIs for each task and generated APIs satisfy the specified constraints.

The formal semantics of the OSEK/VDX standard in $\mathbb{K}$ can also be used for generating test cases. The basic idea is as follows. After a configuration is loaded by the $\mathbb{K}$ tool, a task cell is instantiated for each task according to their corresponding setting in the configuration and is in the *suspended* state. The one whose `AUTOSTART` property is true becomes ready and then is scheduled to run. All the tasks do not have any API to execute initially. We define a set of $\mathbb{K}$ rules which randomly generate an API for the currently running task based on the state of the task. The generated API is then executed based on its formal semantics that is predefined, and the state of the running task is changed correspondingly.

Generated APIs must satisfy some built-in constraints. These constraints are specified in $\mathbb{K}$ rules. For instance, if the API to be executed is `ReleaseResource`, one constraint is that the parameter of the API must be the resource which is the latest one allocated to the task, i.e., the first resource `R` of the sequence in the cell taskResources as shown in the following rule.

```
rule <task> <tid> I </tid> <state> running </state> <apiData> . </apiData>
  <k> . => ReleaseResource(R); </k> <taskRes> ListItem(R) ... </taskRes>
  ... </task> <tcgMode> true </tcgMode> <signal> . </signal>  [transition]
```

The rule also represents three conditions when API can be generated, i.e., the system is running in the test case generation mode as indicated by the cell tcgMode, the running task is undefined as indicated by an empty cell apiData, and no signal is waiting for handling as indicated by the empty cell signal.

If the randomly generated API is `TerminateTask` or `ChainTask`, a sequence of APIs are completed for the running task. That is because there is a constraint that `TerminateTask` and `ChainTask` must be the last API in a task. After the API is executed, the task is terminated and the scheduler selects another task to run according to the scheduling policy. After each task has a generated sequence of APIs, a test case is generated.

An an example, we explain how to generate test cases with the configuration in Fig. 6. Each test case consists of four sequences of APIs for the tasks in the configuration. We feed the configuration into the $\mathbb{K}$ tool and a pattern to which expected results should match, such as the number of APIs for each task. There are two optional parameters specifying the maximal searching depth and the number of test cases. The tool outputs $\mathbb{K}$ configurations that match the specified pattern. In each configurations, every task is assigned with a sequence of APIs, constituting a test case.

The number of test cases may be infinite. For instance, a task can infinitely repeat the process of getting and releasing a resource, making the process of test case generation non-terminating. To solve this problem, We can solve this problem by setting upper bounds to the number of generated test cases and the number of APIs in each task respectively. In the experiment, we defined two patterns denoted by A and B, specifying that in each generated test case there

Table 1. Experimental result of generating two classes of test cases with pattern A (the left table) and B (the right table).

Depth	Solution	Test case	Time (sec)
16	0	0	6
17	92	23	10
20	1024	132	30
21	1468	163	43
22	1748*	200*	65

Depth	Solution	Test cases	Time (sec)
20	0	0	16
21	176	44	23
25	1320	186	76
26	1572	209	97
27	1736*	234*	146

must be at least two tasks which have exactly two and three APIs, respectively. The experimental result is shown in Table 1. Solutions are the configurations returned by $\mathbb{K}$, and they match the specified pattern. We remove the duplicate configurations among them and obtain the number of test cases. The numbers with $*$ mean that they are the upper bound at the corresponding depth. The $\mathbb{K}$ tool runs out of memory once the upper bound exceeds that numbers.

7 Related Work

There are several formalizations of the OSEK/VDX standard. In [13], the semantics of APIs in the standard is formalized using Hoare-logic. The purpose is to verify the correctness of the APIs that are implemented in concrete OSEK/VDX-based operating systems, which is different from ours. The semantics of the standard is formalized using Promela in [7] and NuSMV in [5], with the purpose of test case generation. Compared with their formalization, our formal semantics in $\mathbb{K}$ is more flexible and generic in that they have to assume the list of ready tasks in system is finite because the languages requires the system specified must be of finite-state, while in our formalization we do not have that restriction. Their formalization also requires extra effort to be instantiated according to concrete user-defined applications, while our formalization directly accepts user-defined applications as input without any transformation. Their approaches to test case generation are different from the one described in this paper. For instance, in [5] they use automata to control what is the next possible API to call based on user-given constraints. This can improve the efficiency by avoiding unnecessary trial of those APIs that violate the constraints if they are called. We have tried implementing the automata-based approach in Maude and evaluated that Maude can be used as an efficient test case generator [4]. Since Maude is the underlying rewrite engine of $\mathbb{K}$, we believe that this approach can also be implemented in $\mathbb{K}$ based on our formal semantics of the OSEK/VDX standard.

The OSEK/VDX standard is also formalized using Event-B in [14], and CafeOBJ in [15]. However, in their work they do not explain how to use their formalizations for verification. Our work shows that the formalization of the standard in $\mathbb{K}$ can be effectively used for the verification of concrete OSEK/VDX-based applications and operating systems.

8 Conclusion

We have presented a formal semantics of the OSEK/VDX standard using $\mathbb{K}$, and demonstrated two applications of using the formal semantics to the verification of OSEK/VDX-based applications by symbolic execution and to test case generation. Compared with the existing formalization of the standard, the formal semantics in $\mathbb{K}$ is more flexible and generic in that there is no restriction to the number of tasks, resources and events in the formalization, and it does not need extra effort to instantiate the formal semantics with user-defined applications. Another advantage of the formal semantics is that it can be integrated with the semantics of other prevalent programming languages such as C in order to verify OSEK/VDX-based applications which are implemented in those languages.

References

1. OSEK Group, et al.: OSEK/VDX Operating System Specification (2009)
2. John, D.: OSEK/VDX Conformance Testing-MODISTARC. In: Proceedings of OSEK/VDX Open Systems in Automotive Networks, IET (1998)
3. Zhang, H., Aoki, T., Yatake, K., Zhang, M., Lin, H.H.: An approach for checking OSEK/VDX applications. In: Proceedings of the 13th QSIC, IEEE CSP (2013) 113–116
4. Choi, Y., Zhang, M., Ogata, K.: Evaluation of Maude as a test generation engine for automotive operating systems. (2013) 1–15 Manuscript.
5. Choi, Y.: Constraint Specification and Test Generation for OSEK/VDX-Based Operating Systems. In: Proceedings of the 11th SEFM. Springer (2013) 305–319
6. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Log. Algebr. Program. **79** (2010) 397–434
7. Yatake, K., Aoki, T.: Automatic generation of model checking scripts based on environment modeling. In: Proceedings of the 17th SPIN workshop. Springer (2010) 58–75
8. Zahir, A.: OIL-OSEK implementation language. In: OSEK/VDX Open Systems in Automotive Networks (Ref. No. 1998/523), IEE Seminar, IET (1998) 1–8
9. Clavel, M., Durán, F., et al.: All about Maude. LNCS 4350, Springer (2007)
10. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: Proceedings of the 39th POPL, ACM (2012) 533–544
11. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebr. Program. **81** (2012) 721–781
12. Roşu, G., Şerbănuţă, T.F.: K overview and simple case study. In: Proceedings of International K Workshop (K'11). ENTCS, Elsevier (2013) To appear.
13. Huang, Y., Zhao, Y., Zhu, L., Li, Q., Zhu, H., Shi, J.: Modeling and verifying the code-level osek/vdx operating system with csp. In: Proceedings of the 5th TASE, IEEE CSP (2011) 142–149
14. Vu, D.H., Aoki, T.: Faithfully formalizing OSEK/VDX operating system specification. In: Proceedings of the 3rd SoICT, ACM (2012) 13–20
15. Yatsu, H., Ando, T., Kong, W., Hisazumi, K., Fukuda, A., Aoki, T., Futatsugi, K.: Towards formal description of standards for automotive operating systems. In: Proceedings of 6th ICSTW, IEEE CSP (2013) 13–14