Vlad Rusu (Ed.)

LNCS 11152

Rewriting Logic and Its Applications

12th International Workshop, WRLA 2018
Held as a Satellite Event of ETAPS
Thessaloniki, Greece, June 14–15, 2018
Proceedings

Lecture Notes in Computer Science 11152

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, Lancaster, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Zurich, Switzerland
John C. Mitchell
Stanford University, Stanford, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
C. Pandu Rangan
Indian Institute of Technology Madras, Chennai, India
Bernhard Steffen
TU Dortmund University, Dortmund, Germany
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbrücken, Germany

More information about this series at http://www.springer.com/series/7407

Vlad Rusu (Ed.)

Rewriting Logic and Its Applications

12th International Workshop, WRLA 2018
Held as a Satellite Event of ETAPS
Thessaloniki, Greece, June 14–15, 2018
Proceedings

Springer

Editor
Vlad Rusu
Inria
Lille
France

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-99839-8 ISBN 978-3-319-99840-4 (eBook)
https://doi.org/10.1007/978-3-319-99840-4

Library of Congress Control Number: 2018952569

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

© Springer Nature Switzerland AG 2018
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

This volume contains the papers presented at WRLA 2018: the 12th International Workshop on Rewriting Logic and its Applications, held during June 14–15, 2018, in Thessaloniki, Greece.

There were 21 submissions. Each submission received at least three reviews. The Program Committee decided to accept 12 papers. The program also included three invited talks.

Rewriting 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 domains. It also has good properties as a metalogical framework for representing logics. Several successful languages based on rewriting (ASF+SDF, CafeOBJ, ELAN, Maude) have been designed and implemented.

The aim of WRLA is to bring together researchers with a common interest in rewriting and its applications, and to give them the opportunity to present recent work, discuss future research directions, and exchange ideas.

The topics of the workshop include, but are not limited to:

A. Foundations

- Foundations and models of rewriting and rewriting logic, including termination, confluence, coherence and complexity
- Unification, generalization, narrowing, and partial evaluation
- Constrained rewriting and symbolic algebra
- Graph rewriting
- Tree automata
- Rewriting strategies
- Rewriting-based calculi and explicit substitution

B. Rewriting as a logical and semantic framework

- Uses of rewriting and rewriting logic as a logical framework, including deduction modulo
- Uses of rewriting as a semantic framework for programming language semantics
- Rewriting semantics of concurrency models, distributed systems, and network protocols
- Rewriting semantics of real-time, hybrid, and probabilistic systems
- Uses of rewriting for compilation and language transformation

C. Rewriting languages

- Rewriting-based declarative languages
- Type systems for rewriting

- Implementation techniques
- Tools supporting rewriting languages

D. Verification techniques

- Verification of confluence, termination, coherence, sufficient completeness, and related properties
- Temporal, modal and reachability logics for verifying dynamic properties of rewrite theories
- Explicit-state and symbolic model checking techniques for verification of rewrite theories
- Rewriting-based theorem proving, including (co)inductive theorem proving
- Rewriting-based constraint solving and satisfiability
- Rewriting-semantics-based verification and analysis of programs

E. Applications

- Applications in logic, mathematics, physics, and biology
- Rewriting models of biology, chemistry, and membrane systems
- Security specification and verification
- Applications to distributed, network, mobile, and cloud computing
- Specification and verification of real-time, hybrid, probabilistic, and cyber-physical systems
- Specification and verification of critical systems
- Applications to model-based software engineering
- Applications to engineering and planning

July 2018 Vlad Rusu

Organization

Program Committee

Kyungmin Bae	Pohang University of Science and Technology (POSTECH), South Korea
Roberto Bruni	Università di Pisa, Italy
Stefan Ciobaca	Alexandru Ioan Cuza University, Romania
Francisco Durán	University of Màlaga, Spain
Steven Eker	SRI International, USA
Santiago Escobar	Universitat Politècnica de València, Spain
Maribel Fernandez	KCL
Thomas Genet	IRISA, Rennes, France
Jürgen Giesl	RWTH Aachen University, Germany
Deepak Kapur	University of New Mexico, USA
Helene Kirchner	Inria, France
Alexander Knapp	Universität Augsburg, Germany
Alberto Lluch Lafuente	Technical University of Denmark
Dorel Lucanu	Alexandru Ioan Cuza University, Romania
Salvador Lucas	Universitat Politècnica de València, Spain
Narciso Marti-Oliet	Universidad Complutense de Madrid, Spain
Ugo Montanari	Università di Pisa, Italy
Pierre-Etienne Moreau	Inria-LORIA Nancy, France
Vivek Nigam	Universidade Federal da Paraíba, Brazil
Kazuhiro Ogata	JAIST, Japan
Christophe Ringeissen	Inria, France
Grigore Rosu	University of Illinois at Urbana-Champaign, USA
Vlad Rusu	Inria, France
Ralf Sasse	ETH Zurich, Switzerland
Traian Florin Serbanuta	University of Bucharest, Romania
Mark-Oliver Stehr	SRI International, USA
Carolyn Talcott	SRI International, USA
Martin Wirsing	Ludwig Maximilian University of Munich, Germany
Peter Ölveczky	University of Oslo, Norway

Additional Reviewers

Arusoaie, Andrei
de Carvalho Segundo, Washington
Fuhs, Carsten
Heam, Pierre-Cyrille
Marshall, Andrew M.
Milazzo, Paolo
Pena, Lucas
Sobocinski, Pawel
Vandin, Andrea

Automata and Equations Based Approximations for Reachability Analysis (Invited Talk)

Thomas Genet

Univ Rennes/Inria/CNRS/IRISA, Campus Beaulieu,
35042 Rennes Cedex, France

Term Rewriting Systems (TRSs for short) are a convenient formal model for software systems. This formalism is expressive enough to model in a simple and accurate way many aspects of computation such as: recursivity, non-determinism, parallelism, distribution, communication. On such models, verification is facilitated by the large collection of proof techniques of rewriting: termination, non-termination, confluence, non-confluence, reachability, unreachability, inductive properties, etc. This talk focuses on unreachability properties of a TRS, which entails safety properties on the modeled software system.

Starting from a single term s, proving that t is unreachable, i.e., $s \not\to^*_{\mathcal{R}} t$ is straightforward if $\mathcal{R}$ is terminating. This problem is undecidable if $\mathcal{R}$ is not terminating or if we consider infinite sets of initial terms s and infinite sets of "Bad" terms t. There exists TRSs classes for which those problems are decidable. For those classes, decidability comes from the fact that the set of reachable terms is *regular*, i.e., it can be recognized by a tree automaton [5]. Those classes are surveyed in [7].

However, TRSs modeling software systems do not belong to those "decidable classes", in general. The rewriting and tree automata community have proposed different techniques to over-approximate the set of reachable terms. Over-approximating reachable terms provide a criterion for unreachability on TRSs and, thus, a criterion for safety of the modeled systems. Those approximation techniques range from TRSs transformation [11], ad hoc automata transformations [3, 6, 10], CounterExample-Guided Abstraction Refinement (CEGAR) [1, 2, 4], and abstraction by equational theories [9, 12]. I will present the principles underlying those techniques, discuss their pros and cons, and recall some of their applications. Then, I will present a recent attempt to combine abstraction by equational theories and CEGAR to infer accurate over-approximations for TRSs modeling higher-order functional programs [8].

References

1. Boichut, Y., Boyer, B., Genet, T., Legay, A.: equational abstraction refinement for certified tree regular model checking. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635. Springer, Heidelberg (2012)

2. Boichut, Y., Courbis, R., Héam, P.C., Kouchnarenko, O.: Finer is better: abstraction refinement for rewriting approximations. In: Voronkov, A. (eds.) RTA 2008. LNCS, vol. 5117, pp. 48–62. Springer, Heidelberg (2008)
3. Boichut, Y., Héam, P.-C., Kouchnarenko, O.: Automatic approximation for the verification of cryptographic protocols. In: Proceedings of AVIS 2004. Joint to ETAPS 2004, Barcelona, Spain (2004)
4. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking. ENTCS, **149**(1), 37–48 (2006)
5. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., Tommasi, M.: Tree automata techniques and applications (2008). http://tata.gforge.inria.fr
6. Genet, T.: Decidable approximations of sets of descendants and sets of normal forms. In: Nipkow, T. (eds.) RTA 1998. LNCS, vol 1379, pp. 151–165. Springer, Heidelberg (1998)
7. Genet, T.: Reachability analysis of rewriting for software verification. Habilitation document, Université de Rennes 1 (2009). http://people.irisa.fr/Thomas.Genet/publications.html
8. Genet, T., Haudebourg, T., Jensen, T.: Verifying higher-order functions with tree automata. In: Baier, C., Dal Lago, U. (eds.) FoSSaCS 2018. LNCS, vol. 10803, Springer, Cham (2018, to be published)
9. Genet, T., Rusu, R.: Equational tree automata completion. J. Symb. Comput. **45**, 574–597 (2010)
10. Genet, T., Tong, V.V.T.: Reachability analysis of term rewriting systems with timbuk. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNAI, vol. 2250, pp. 691–702, Springer-Verlag (2001)
11. Jacquemard, F.: Decidable approximations of term rewriting systems. In: Ganzinger, H. (eds.) RTA 1996. LNCS, Vol. 1103, pp. 362–376, Springer, Heidelberg (1996)
12. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. TCS, **403**(2–3), 239–264 (2008)

Contents

Benchmarking Implementations of Term Rewriting and Pattern Matching in Algebraic, Functional, and Object-Oriented Languages: The 4th Rewrite Engines Competition 1
Hubert Garavel, Mohammad-Ali Tabikh, and Imad-Seddik Arrada

Multi-paradigm Programming in Maude 26
Santiago Escobar

MUnit: A Unit Framework for Maude 45
Adrián Riesco

Parameterized Programming for Compositional System Specification 59
Óscar Martín, Alberto Verdejo, and Narciso Martí-Oliet

Symbolic Specification and Verification of Data-Aware BPMN Processes Using Rewriting Modulo SMT 76
Francisco Durán, Camilo Rocha, and Gwen Salaün

Associative Unification and Symbolic Reasoning Modulo Associativity in Maude 98
Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott

Proving Structural Properties of Sequent Systems in Rewriting Logic 115
Carlos Olarte, Elaine Pimentel, and Camilo Rocha

Formal Modeling and Analysis of the Walter Transactional Data Store 136
Si Liu, Peter Csaba Ölveczky, Qi Wang, and José Meseguer

Extending Timbuk to Verify Functional Programs. 153
Thomas Genet, Tristan Gillard, Timothée Haudebourg, and Sébastien Lê Cong

Generalized Rewrite Theories and Coherence Completion 164
José Meseguer

Proving Ground Confluence of Equational Specifications Modulo Axioms . . . 184
Francisco Durán, José Meseguer, and Camilo Rocha

Uniform Strong Normalization for Multi-discipline Calculi. 205
Paul Downen, Philip Johnson-Freyd, and Zena M. Ariola

Real-Time Rewriting Logic Semantics for Spatial Concurrent
Constraint Programming 226
Sergio Ramírez, Miguel Romero, Camilo Rocha, and Frank Valencia

Approximating Any Logic Program by a CS-Program 245
Yohan Boichut, Vivien Pelletier, and Pierre Réty

Author Index 261

Benchmarking Implementations of Term Rewriting and Pattern Matching in Algebraic, Functional, and Object-Oriented Languages

The 4th Rewrite Engines Competition

Hubert Garavel(✉), Mohammad-Ali Tabikh, and Imad-Seddik Arrada

Univ. Grenoble Alpes, INRIA, CNRS, LIG, 38000 Grenoble, France
hubert.garavel@inria.fr

Abstract. Many specification and programming languages have adopted term rewriting and pattern matching as a key feature. However, implementation techniques and observed performance greatly vary across languages and tools. To provide for an objective comparison, we developed an open, experimental platform based upon the ideas of the three Rewrite Engines Competitions (2006, 2008, and 2010), which we significantly enhanced, extended, and automated. We used this platform to benchmark interpreters and compilers for a number of algebraic, functional, and object-oriented languages, and we report about the results obtained for CafeOBJ, Clean, Haskell, LNT, LOTOS, Maude, mCRL2, OCaml, Opal, Rascal, Scala, SML (MLton and SML-NJ), Stratego/XT, and Tom.

Keywords: Abstract data type · Algebraic specification · Compiler
Functional programming · Interpreter · Object-oriented programming
Programming language · Specification language · Term rewrite engine
Term rewrite system

1 Introduction

There is a large corpus of scientific work on term rewriting. Beyond theoretical results, one main practical result is the introduction of algebraic terms and rewrite rules into many specification and programming languages, some of which are no longer available or maintained, some of which remain confidential, but a few others are widespread or increasingly popular.

In some of these languages, one directly finds the familiar concepts of abstract sorts, algebraic operations, and equations or rewrite rules to define the semantics of operations. In other languages based on functional, imperative, or object-oriented paradigms, one finds the term rewriting concepts under derived/restricted forms: types inductively defined using free constructors, and pattern matching.

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 1–25, 2018.
https://doi.org/10.1007/978-3-319-99840-4_1

In the presence of many different languages, which one(s) should be preferred for concrete applications? The present study addresses this question by assessing the performance of languages implementations on a set of rewriting-oriented benchmarks. A decade ago, this question was already dealt with by the three Rewrite Engines Competitions (REC, for short), organized in 2006 [8], 2008 [12], and 2010 [11]. Our work builds upon these three competitions.

A more personal motivation for undertaking the present study was related to CÆSAR.ADT [14,20], a compiler developed at INRIA Grenoble since the late 80s. Based on Schnoebelen's pattern-matching compiling algorithm [41], CÆSAR.ADT translates LOTOS abstract data types (seen as many-sorted conditional term rewrite systems with free constructors and priorities between equations) into C code. In 1992, CÆSAR.ADT, initially written in C, was rewritten to a large extent in LOTOS abstract data types — it might have been the first rewriting engine to bootstrap itself, slightly before Opal [10] and long before ASF+SDF [4]. Since then, CÆSAR.ADT has been routinely used as part of the CADP toolbox [17] for model checking specifications of concurrent systems. For this purpose, it has been specifically optimized to reduce memory consumption by implementing data structures very compactly. It has also been used to build two large compilers: itself (using bootstrapping) and the XTL compiler [31].

In 2007, a comparative study [44] reported average performance results for CÆSAR.ADT, but only on a small number of benchmarks. This triggered our desire to assess CÆSAR.ADT against widespread implementations of term rewriting and pattern matching, to learn if this compiler is still state of the art.

Another personal motivation behind the present study concerns LNT [18], which has been designed as a user-friendly replacement language for LOTOS, more suitable for use in industry. These two languages are quite different: LOTOS relies upon ACT-ONE's [13] abstract data types, whereas LNT is a clean combination of imperative-programming constructs with first-order functional-programming discipline: mutable variables, assignments, `if-then-else` conditionals, pattern-matching `case`, `while` and `for` loops with `break`, functions having `return` statements and `in`, `out`, and `in-out` parameters, etc.

At the moment, definitions of LNT types and functions are implemented in two successive steps: first, by translation [18] to LOTOS abstract data types, using a generalization of the ideas proposed in [40], then to C code, by reusing the aforementioned CÆSAR.ADT compiler. This is a quite challenging approach, since a language with an imperative syntax (LNT) is first translated to term rewrite systems, then back to another imperative language (C). One may thus wonder whether such a two-step translation, dictated by software-reuse considerations, is efficient enough in practice, especially for model-checking verification [17], which is highly demanding in terms of memory and computing time.

Assessing the performance of implementations of term rewriting and pattern matching (including the case of LOTOS and LNT) raises a number of difficult questions. Is it possible to compare very different languages on an equal footing? Which are the right tools to be involved in the comparison? Where can one

find the term rewriting specifications to be used as benchmarks? What is the experimental setting suitable for a proper assessment?

The present article addresses these questions. It is organized as follows. Section 2 lists the various tools implementing term rewriting and pattern matching we considered for this study. Section 3 describes the common language REC-2017 in which benchmarks (i.e., conditional term rewrite systems) can be encoded. Section 4 presents the translators we developed for converting this common language to the input languages of the tools. Section 5 reports about the collection of 85 benchmarks prepared for our study. Taking advantage of this collection, Sect. 6 provides quantified insight about the verbosity of the input languages. Section 7 describes the execution platform used to run the tools on the benchmarks. Section 8 gives experimental results and draws the Top-5 podium of the most efficient tools. Section 9 discusses potential threats to validity. Finally, Sect. 10 provides concluding remarks.

2 Selected Tools

Table 1 lists all the languages assessed by the present study. For each language, column 2 gives bibliographic references; column 3 indicates whether the language is *alg*ebraic, *fun*ctional, or *obj*ect-oriented[1]; column 4 gives the name of the tool (interpreter or compiler) used to execute programs written in this language and the version number of this tool[2]; column 5 gives the URL of the reference web site.

The present study assesses numerous languages/tools that were neither considered in [44] nor in the three Rewrite Engines Competitions [8,11,12]: CafeOBJ, LNT, LOTOS, OCaml, Opal, Rascal, Scala, and SML. Conversely, a few languages/tools assessed during these former studies have not been retained for the present study: ASF+SDF (superseded by Rascal and Stratego/XT), Elan (superseded by Tom), μCRL (superseded by mCRL2), Termware (its last version 2.3.3 was issued in May 2009 and the tool did not participate in the 3rd Competition), and TXL (the developer informed us that the tool was not designed for the REC benchmarks and that there was no point in trying it). Concerning the languages/tools listed in Table 1, the following remarks can be made:

- Certain languages (namely, OCaml and Rascal) possess both an interpreter and a compiler; we evaluated each of them.
- The mCRL2 tool set provides two different rewriting engines: *jitty* (*just-in-time* [38,39]) and *jittyc* (*just-in-time compiled*), which we both evaluated. The *innermost* rewriter mentioned in [44] is no longer available.

[1] This classification is subjective, since some languages belong to multiple paradigms: LNT claims to be functional and imperative, Maude algebraic and object-oriented, Opal algebraic and functional, OCaml and Scala functional and object-oriented, etc.; due to lack of space, we only indicate the paradigm that we consider to be the principal one, within the scope of this study.

[2] The tool name is only mentioned if it is different from the language name.

Table 1. List of tools considered for the 4th Rewrite Engines Competition

Language	Bib. ref.	Kind	Tool version	Web site
CafeOBJ	[9]	alg.	1.5.5	http://cafeobj.org
Clean	[37]	fun.	2.4 (May 2017)	http://clean.cs.ru.nl
Haskell	[30]	fun.	GHC 8.0.1	http://www.haskell.org
LNT	[6,17]	fun.	CADP 2017-b	http://cadp.inria.fr
LOTOS	[17,24]	alg.	CADP 2017-b	http://cadp.inria.fr
Maude	[7]	alg.	2.7.1	http://maude.cs.illinois.edu
mCRL2	[22]	alg.	201409.0	http://www.mcrl2.org
OCaml	[28]	fun.	4.04.1	http://www.ocaml.org
Opal	[36]	alg.	OCS 2.4b	http://projects.uebb.tu-berlin.de/opal
Rascal	[3]	obj.	0.8.0	http://www.rascal-mpl.org
Scala	[35]	obj.	2.11.8	http://www.scala-lang.org
SML	[34]	fun.	MLton 20130715	http://www.mlton.org
SML	[34]	fun.	SML/NJ 110.80	http://www.smlnj.org
Stratego[a]	[5]	alg.	2.1.0	http://www.metaborg.org
Tom	[1]	obj.	2.10	http://tom.loria.fr

[a]The full name is “Stratego/XT”, which we often abbreviate to “Stratego” so as to save space in tables

- SML (i.e., Standard ML) is the only language in Table 1 that does not allow conditional patterns. To overcome this limitation, we chose to use SML jointly with the Nowhere preprocessor [29] that adds support for Boolean guards in pattern matching.
- Among the many SML implementations available, we selected two compilers: SML/NJ and MLton.
- We evaluated the Clean compiler in two different ways: (i) by invoking the compiler `clm` in the standard way; (ii) by designing a custom script `clm-hack` that invokes `cocl` and `clm` with special options. The latter approach was suggested by the developers of Clean to address performance issues.

3 The REC-2017 Language

For benchmarking implementations of term rewriting, a crucial difficulty is that almost all the tools listed in Sect. 2 have a different input language — the only exception being SML/NJ and MLton, which both operate on SML programs. To address this issue, the organizers of the 2nd and 3rd Rewrite Engines Competition designed a common language [12, Sect. 3] [11, Sect. 3.1], which they named REC and to which we will refer as REC-2008. It is a human-readable, tool-independent format for describing many-sorted conditional term rewrite systems, as well as property queries (namely, confluence checks and computation

of normal forms) on such systems. The main limitation of REC-2008 was its exclusive focus on algebraic specification languages.

We adopted REC-2008 as a starting point, but gradually evolved it to make it compatible with the characteristics of functional and object-oriented languages. Such evolution led to a new language REC-2017[3], which is defined by the EBNF grammar of Table 2, in which ⟨*rec-spec*⟩ is the axiom and where $\langle e\rangle^{*[c]}$ (respectively, $\langle e\rangle^{+[c]}$) denotes the concatenation of $n \geq 0$ (resp. $n > 0$) non-terminals ⟨*e*⟩ separated by the character *c*, which can be a space, a comma, or a newline (noted "`\n`"). Semicolons can be used wherever commas can be used, which is sometimes convenient for structuring long lists of parameters (see, e.g., [19]). A simple example of a REC-2017 specification is given in Table 3.

Contrary to REC-2008, REC-2017 is strictly line-based, meaning that newline characters may only occur where they are specified in the grammar. Comments start with `%` and extend to the end of the line. Inclusion of external files is possible using `#include` directives similar to those that exist in the C language.

Identifiers (i.e., non-terminals ⟨*spec-id*⟩, ⟨*sort-id*⟩, ⟨*cons-id*⟩, ⟨*opn-id*⟩, and ⟨*var-id*⟩) always start with a letter, followed by any number of letters, digits, `_` (underscores), `'` (primes), or `"` (seconds); thus, fancy notations for numbers and infix functions (e.g., `123`, `+`, `&`, `==`, etc.) are not supported. Identifiers must be different from the grammar keywords (i.e., `SORTS`, `CONS`, `if`, etc.) and must

Table 2. EBNF grammar defining the syntax of the REC-2017 language

⟨*rec-spec*⟩ ::= `REC-SPEC` ⟨*spec-id*⟩ `\n`	⟨*rule*⟩ ::= ⟨*left*⟩ `->` ⟨*term*⟩
`SORTS` `\n`	| ⟨*left*⟩ `->` ⟨*term*⟩ `if` ⟨*cond*⟩
$\langle$*sort-id*$\rangle^{*[\ ]}$ `\n`	⟨*left*⟩ ::= ⟨*opn-id*⟩
`CONS` `\n`	| ⟨*opn-id*⟩ `(`$\langle$*pattern*$\rangle^{+[,]}$`)`
$\langle$*cons-decl*$\rangle^{*[\backslash n]}$ `\n`	⟨*pattern*⟩ ::= ⟨*var-id*⟩
`OPNS` `\n`	| ⟨*cons-id*⟩
$\langle$*opn-decl*$\rangle^{*[\backslash n]}$ `\n`	| ⟨*cons-id*⟩ `(`$\langle$*pattern*$\rangle^{+[,]}$`)`
`VARS` `\n`	⟨*term*⟩ ::= ⟨*var-id*⟩
$\langle$*var-decl*$\rangle^{*[\backslash n]}$ `\n`	| ⟨*cons-id*⟩
`RULES` `\n`	| ⟨*cons-id*⟩ `(`$\langle$*term*$\rangle^{+[,]}$`)`
$\langle$*rule*$\rangle^{*[\backslash n]}$ `\n`	| ⟨*opn-id*⟩
`EVAL` `\n`	| ⟨*opn-id*⟩ `(`$\langle$*term*$\rangle^{+[,]}$`)`
$\langle$*term*$\rangle^{*[\backslash n]}$ `\n`	⟨*cond*⟩ ::= ⟨*term*⟩ `-><-` ⟨*term*⟩
`END-SPEC` `\n`	| ⟨*term*⟩ `->/<-` ⟨*term*⟩
	| ⟨*cond*⟩ `and-if` ⟨*cond*⟩
⟨*cons-decl*⟩ ::= ⟨*cons-id*⟩ `:` $\langle$*sort-id*$\rangle^{*[,]}$ `->` ⟨*sort-id*⟩	
⟨*opn-decl*⟩ ::= ⟨*opn-id*⟩ `:` $\langle$*sort-id*$\rangle^{*[,]}$ `->` ⟨*sort-id*⟩	
⟨*var-decl*⟩ ::= $\langle$*var-id*$\rangle^{*[\ ]}$ `:` ⟨*sort-id*⟩	

[3] See also http://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/doc.

Table 3. Simple REC-2017 specification defining Booleans and natural numbers

```
REC-SPEC simple
SORTS    % abstract data domains
  Bool Nat
CONS    % primitive operations
  true : -> Bool
  false : -> Bool
  zero : -> Nat
  succ : Nat -> Nat
OPNS    % defined functions
  and : Bool Bool -> Bool
  plus : Nat Nat -> Nat
VARS    % free variables
  A B : Bool
  M N : Nat
RULES    % function definitions
  and (A, B) -> B if A -><- true
  and (A, B) -> false if A -><- false
  plus (zero, N) -> N
  plus (succ (M), N) -> succ (plus (M, N))
EVAL    % terms to be evaluated
  and (true, false)
  plus (succ (zero), succ (zero))
END-SPEC
```

not be prefixed with `REC_`, `rec_`, `Rec_`, or any case-sensitive variation of these. There is a unique name space for all identifiers, meaning, for instance, that it is not possible to have a sort and a variable sharing the same identifier. As in REC-2008, operation overloading is not permitted. Also, two identifiers must not differ only by their case (e.g., defining `x0` and `X0` simultaneously is invalid).

A REC-2017 specification is divided into six sections. The `SORTS` section declares a set of sorts. The `CONS` and `OPNS` sections respectively declare two sets of constructor and non-constructor operations; each operation is given by its identifier, the (possibly empty) list of sorts of its arguments and the sort of its result. The `VARS` section declares a set of free variables. The `RULES` section contains a list of rewrite rules; the left-hand side of each rule defines a non-constructor on arguments specified by a pattern (i.e., a term containing only constructors and free variables), whereas the right-hand side may be an arbitrary term; rewrite rules may be conditional or not; conditions, if present, are the logical conjunction (noted `and-if`) of one or many elementary conditions of the form "t_1`-><-`t_2" (meaning that both terms t_1 and t_2, which have the same sort,

can be rewritten to some common term t) or "t_1`->/<-`t_2" (which is the negation of "t_1`-><-`t_2"). The `EVAL` section gives a list of closed terms (i.e., terms that do not contain free variables), the ground normal forms of which have to be computed.

There are additional static semantic constraints: in the `CONS` section, constructors having the same result sort must be gathered and declared in sequence; in the `OPNS` section, non-constructors with arity zero (i.e., constants) must be declared before being used[4]; each pattern and each term present in a REC-2017 specification must be well-typed; implicit type conversions between different sorts are not allowed; in the `RULES` section, all variables used in rewrite rules must have been declared in the `VARS` section; each variable occurring on the right-hand side of a rewrite rule (including in conditions) must be present in the left-hand side pattern of this rule, i.e., the REC-2017 specification is a 1-CTRS according to the classification proposed in [33, Definition 6.1]; each left-hand side of a rule must be linear, i.e., must not contain multiple occurrences of the same variable[5]; the set of rules defining the same non-constructor (i.e., all rules whose left-hand sides have the same ⟨*opn-id*⟩) must be gathered and appear in sequence; these rules must be deterministic, meaning that either their left-hand side patterns are disjoint[6] or their conditions (if any) are mutually exclusive, thus implying confluence; these rules do not have to be complete, meaning that partial functions are allowed, provided they are invoked only on arguments for which they are defined; finally, termination is also required, even if it is not always decidable. Hence, because the specification is convergent, all tools, whichever rewrite strategy they implement, should terminate and produce the same results when evaluating the terms listed in the `EVAL` section.

Compared to REC-2008, the main change brought by REC-2017 is the distinction between constructors and non-constructors: in REC-2008, all operations were declared in one single section `OPS`, whereas REC-2017 introduces two separate sections `CONS` and `OPNS`, as well as the free-constructor discipline, i.e., the prohibition of equations between constructors implied by the syntactic definitions of ⟨*left*⟩ and ⟨*pattern*⟩ in Table 2. The "`get normal form for`" directives of REC-2008 have been replaced by the `EVAL` section of REC-2017. Also, some features of REC-2008 that were only used in some algebraic languages and have no counterpart in functional and object-oriented languages have also been removed, namely, the "`check the confluence of`" query and the OBJ/Maude-like `assoc` (associativity), `comm` (commutativity), `id` (identity, i.e., neutral element), and `strat` (strategy) attributes.

4 The REC-2017 Translators

Following the introduction of the REC-2008 language, various approaches were adopted during the 2nd and 3rd Rewrite Engines Competitions, to process

[4] This constraint arises from OCaml and SML; it eases the translation from REC-2017 to these languages.

[5] Non-linear patterns can be replaced by linear patterns by adding extra conditions.

[6] Consequently, there is no notion of priority between rewrite rules.

specifications written in this language: (i) Maude was enriched with an environment to parse REC-2008 specifications; (ii) translators were developed to convert REC-2008 into the input languages accepted by ASF+SDF, Stratego/XT, and Tom; unfortunately, these translators are no longer available today or their target languages have evolved; (iii) in other cases, the translation from REC-2008 to the input languages of the remaining tools (e.g., TXL) had to be done manually.

Our study is significantly broader in scope, as we are considering many more input languages, and we also plan to have more benchmarks written in REC-2017. Manually translating these benchmarks into each input language, detecting and correcting the unavoidable mistakes introduced during translation, and maintaining consistency between hundreds or thousands of files on the long run would not have been feasible.

It would not have been realistic either to ask tool developers to modify their tools to directly parse REC-2017 specifications. We therefore undertook the development of automated translators from REC-2017 to the 13 input languages. Actually, we developed 17 translators in total, since we experimented two different translations for both CafeOBJ (noted "CafeOBJ-A" and "CafeOBJ-B")[7] and Tom (noted "Tom-A" and "Tom-B")[8], and we also built a translator that produces files in the TRS input format of the AProVE termination checker[9] [21].

To keep these translators as simple as possible, we made a few radical decisions concerning their design and implementation.

The REC-2017 language has many static semantics constraints (listed in Sect. 3), which need to be checked automatically, since term rewrite systems are an error-prone formalism. Ideally, each benchmark should be checked at the REC-2017 source-code level, before being translated to the various target languages. Instead, we took the reverse approach: no checks are done *before* translation, all checks being done *after* translation. Concretely, each benchmark is translated first and the results of the translators are checked afterwards, thus deferring the verification of static semantics constraints to the target compilers and interpreters. In this approach, a REC-2017 benchmark is deemed to be correct if all its translations are accepted by the corresponding tools. The confluence property is checked by the Opal compiler, which enforces a sufficient condition of determinism, and the termination property is checked by the AProVE tool, which often succeeds in proving quasi-decreasingness, but sometimes loops forever on certain benchmarks.

We chose to perform translation at a mostly syntactic level, excluding all sophisticated semantic optimizations that could advantage or disadvantage certain target tools. This way, all the tools receive semantically-equivalent input files that only differ by syntax. In particular, we decided that the translators should not try to exploit the predefined types (Booleans, integers, polymorphic

[7] CafeOBJ-A uses equations (`eq`, `ceq`, and `red` clauses), whereas CafeOBJ-B uses rewrite rules (`trans`, `ctrans`, and `exec` clauses).

[8] Tom-A makes no difference between constructors and non-constructors, whereas Tom-B defines non-constructors by pattern-matching (`%match` clause).

[9] http://aprove.informatik.rwth-aachen.de.

lists, etc.) and predefined functions (arithmetic operators, etc.) that may exist in the target languages. Concretely, all the target tools receive the constructors attached to each sort and the rewrite rules attached to each non-constructor, such information being unchanged from the source REC-2017 benchmarks. This was dictated by three reasons: (i) keeping translations simple; (ii) avoiding difficult choices between bounded-precision and arbitrary-precision integers; (iii) focusing assessment on the implementation of algebraic terms and rewrite rules.

Rather than using sophisticated compiler-construction tools (such as Rascal, Stratego/XT, or even LNT [16]) to build our translators, we adopted an agile, lightweight approach based on scripting. Taking advantage of the simple syntax of REC-2017 (based upon lines and code sections), we wrote our translators using a combination of `awk` (3070 lines, excluding blank lines and comments) and Bourne shell (1080 lines), with intensive use of Unix commands such as `cpp`, `grep`, and `sed`, all interconnected by pipes in data-flow programming style. Although this approach is not the most commendable nor the most optimal (for instance, the translator to Clean requires four passes), it has the merits of flexibility and conciseness (244 lines per translator on average).

Although all our translators are different, many of them share common traits, which can be summarized by the following list of questions concerning the target languages and the corresponding answers gathered in Table 4:

- Column (a): Are constructors and non-constructors handled differently (noted “D”) or identically (noted “I”)?
- Column (b): Are constructors declared together with their result type (noted “T”) or separately (noted “S”)?
- Column (c): Does the target language provide predefined equality/inequality functions (noted “E”) or must these functions be defined explicitly (noted “–”)?
- Column (d): Does the target language provide a predefined printing function for algebraic terms (noted “P”) or must such a function be defined explicitly, e.g., as a monad (noted “–”)?
- Column (e): Are rewrite rules encapsulated inside the non-constructor function they define (noted “F”) or can they occur separately (noted “S”)?
- Columns (f), (g), (h), and (i): Should type (resp. constructor, non-constructor, free-variable) identifiers start with a lower-case letter (noted “L”), an upper-case letter (noted “U”), or any of these (noted “–”)?
- Columns (j) and (k): Should a constructor (resp. non-constructor) f with arity zero be invoked as “$f\,()$” (noted “()”) or simply as “f” (noted “–”)?
- Columns (l) and (m): Should a constructor (resp. non-constructor) f with arity $n > 0$ be invoked as “$f\,(e_1, ..., e_n)$” (noted “A” for application) or “$f\ e_1\ ...\ e_n$” (noted “J” for juxtaposition)?

Our translators do not build abstract syntax trees when parsing REC-2017 specifications. Instead, they exploit the line-based structure of REC-2017, so that most of the translation is done using substitutions specified by regular expressions. Although some parts of REC-2017 are not regular (namely, the

Table 4. Overview of the idiosyncrasies of all target languages

Language	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)
CafeOBJ-(A,B)	D[a]	S	E	P	S	–	–	–	–	–	–	A	A
Clean	D	T	–[b]	–	S	U	U	L	L	–	–	J	J
Haskell	D	T	E	P	S	U	U	L	L	–	–	J	J
LNT	D	T	E	P	F	–	–	–	–	–[c]	–[c]	A	A
LOTOS	D[d]	S	–	P	S	–	–	–	–	–	–	A	A
Maude	I[e]	S	E	P	S	–	–	–	–	–	–	A	A
mCRL2	D	T	E	P	S	–	–	–	–	–	–	A	A
OCaml	D	T	E	–	F	L	U	L	L	–	–	J	A
Opal	D	T	E	–	S	L	U	L	L	–	–	A	A
Rascal	D	T	E	P	S	U	U	L	L	()	()	A	A
Scala	D	T	E	P	F	U	U	L	L	()	()	A	A
SML	D	T	E	–	F	L	U	L	L	–	()	A	A
Stratego	I	S	–	P	S	–	–	–	–	()	()	A	A
Tom-A	I	T[f]	E	P	S	–	–	–	–	()	()	A	A
Tom-B	D	T	E	P	F	–	–	–	–	()	()	A	A

[a] CafeOBJ has annotations "{`constr`}" for constructors.
[b] In Clean, the library module `GenEq` provides generic comparison functions `===` and `=!=`, but we decided not to use them, as we were informed that they could be less efficient than user-defined comparison functions.
[c] In LNT, empty parentheses are optional after constructors and non-constructors with arity zero.
[d] Standard LOTOS does not have the notion of constructor, but the CÆSAR.ADT compiler introduces a distinction between constructors and non-constructors by means of comments "`(*! constructor *)`".
[e] Maude has annotations "`[ctor]`" that play no role when interpreting rewrite rules, but are understood and used by complementary tools (such as the Maude sufficient completeness checker).
[f] In abstract-syntax definitions for Tom-A, no distinction is made between constructors and non-constructors; they are all declared together with their result type.

sub-languages described by the $\langle pattern \rangle$ and $\langle term \rangle$ non-terminals, which are obviously context-free), regular expressions are sufficient to translate these parts, as the target languages are syntactically close to REC-2017.

The trickiest point is probably the translation of REC-2017 expressions (resp. patterns) written in application form, i.e., "$f\ (e_1, ..., e_n)$", into equivalent expressions (resp. patterns) written in juxtaposition form, i.e., "$f\ e_1\ ...\ e_n$". In most cases, this translation can be done using the Unix command `sed`, by applying two successive substitutions based on regular expressions: first, all commas are removed using the substitution [`,` $\rightarrow \epsilon$]; then, each function symbol f followed by an opening parenthesis is moved after this parenthesis, using the substitution

[f(→ (f]. For instance, the term "`f (a, g (b, c), h (d))`" is first translated to "`f (a g (b c) h (d))`" and then to "`(f a (g b c) (h d))`".

The only exception is OCaml, which has different syntaxes for calling constructors and non-constructors, as shown in Table 5. To address this problem, we equipped our translator to OCaml with a small C program that distinguishes between constructors and non-constructors, and counts the nesting level of parentheses to decide whether commas between arguments must be preserved or removed. More generally, our most involved translators are those for ML-based languages (namely, OCaml and SML) because these languages have many particular syntactic cases (e.g., for constructors and non-constructors with arity zero or one), as well as static-semantics rules that forbid forward declarations (e.g., a constant function has to be declared before it is used) to promote readability.

Table 5. Regularity and irregularity in OCaml syntax

arity	kind	call in REC	declaration in OCaml	call in OCaml
$= 0$	non-constructor	f	**let** $f : t = ...$	f
$= 0$	constructor	C	**type** $t = C \mid ...$	C
$= 1$	non-constructor	$f\,(e_1)$	**let** $f\,(x_1{:}t_1) : t = ...$	$(f\ e_1)$ or $f\,(e_1)$
$= 1$	constructor	$C\,(e_1)$	**type** $t = C$ **of** $t_1 \mid ...$	$(C\ e_1)$ or $C\,(e_1)$
> 1	non-constructor	$f\,(e_1, ..., e_n)$	**let** $f\,(x_1{:}t_1) ... (x_n{:}t_n) : t = ...$	$(f\ e_1\ ...\ e_n)$
> 1	constructor	$C\,(e_1, ..., e_n)$	**type** $t = C$ **of** $t_1 * ... * t_n \mid ...$	$C\ (e_1,\ ...\ ,\ e_n)$

5 Selected Benchmarks

To assess the tools listed in Sect. 2, a collection of conditional term rewrite systems is necessary. Despite the abundant literature on term rewriting, very few benchmarks are available on the Web. We therefore considered with great care the benchmarks developed during the 2006, 2008, and 2010 Rewrite Engines Competitions [8,11,12]. We completed these benchmarks with alternative versions kindly sent to us by F. Durán and P.-E. Moreau. We also added the benchmarks used in [44].[10]

Because these benchmarks have not been written by us, our initial intent was to keep them unchanged as much as possible when translating them into the REC-2017 language, so as to avoid introducing bias in subsequent experiments. However, we progressively realized that most benchmarks could not be reused directly, and that a number of changes were required to make them usable:

– Many benchmarks existed in several variants: we removed duplicated benchmarks, trying to follow differences between original and derived versions, to retain only the best variant.

[10] All these benchmarks are available from http://gforge.inria.fr/scm/viewvc.php/rec/2006-REC1, http://gforge.inria.fr/scm/viewvc.php/rec/2008-REC2, http://gforge.inria.fr/scm/viewvc.php/rec/2010-REC3, and http://gforge.inria.fr/scm/viewvc.php/rec/2007-Weerdenburg.

- A few benchmarks were incorrect. For instance, the Fibonacci suite was improperly defined for $n = 1$. In some other benchmarks, a few operations had been lost during manual translations between different input languages. Specific actions were taken to repair such benchmarks.
- In many benchmarks, deep changes were made to enforce the separation (introduced in the REC-2017 language) between constructors and non-constructors. Constructors were identified and all equations between constructors were removed by splitting each problematic operation into a free constructor and a non-constructor [43]. Similar changes were done to also eliminate the implicit equations between constructors added by the three REC-2008 attributes `assoc` (associativity), `comm` (commutativity), and `id` (neutral element).
- A few benchmarks designed to perform rewriting on open terms (i.e., terms containing free variables) have been modified to rewrite only closed terms in their `EVAL` section.
- Some benchmarks had been specifically written for tools that assume different priority levels among rewrite rules (e.g., certain "default" rules are only applied if all other rules have failed). Such benchmarks were modified by adding extra conditions to these "default" rules, so as to avoid non-termination issues with tools not implementing priorities.
- Similarly, some benchmarks had been written for tools that apply particular rewrite strategies such as lazy evaluation, just-in-time rewriting, etc. Other benchmarks relied on the REC-2008 attribute `strat` that specifies in which order the arguments of an operation must be evaluated. A typical example was the $\texttt{ifthenelse}(c, x, y)$ operation defined by two rules: $\texttt{ifthenelse}(\texttt{true}, x, y) \rightarrow x$ and $\texttt{ifthenelse}(\texttt{false}, x, y) \rightarrow y$, which often caused performance or non-termination issues with rewriting engines based on functional application, as x and y are both evaluated whatever the value of c. Such benchmarks were modified, e.g., by replacing `ifthenelse` operations with conditional rules.
- Confluence was checked by translation to Opal, which requires deterministic rules. A few benchmarks were purposely non confluent, since they intended to compute all solutions to a problem. Two of them (`confluence` and `soundness-of-parallel-engines`) even used an undocumented "`get all normal forms of`" query. Because the REC-2017 language requires rewrite rules to be deterministic, thus confluent, such benchmarks had to be severely restricted; in some cases, their original intent was entirely lost, but this is a price to pay for having benchmarks that can be processed by a majority of tools.
- Termination was checked, whenever possible, by translation to AProVE, which produced proofs of quasi-decreasingness for many benchmarks.
- Certain benchmarks are intrinsically parametric: for instance, computing the factorial $n!$ depends on the value of n; solving the Hanoi tower problem depends on the number of disks; etc. Existing benchmarks would often test several parameter values in the same REC file, with the drawback that one could not distinguish between tools that quickly fail on small parameter

values and tools that succeed on small values, but later fail on larger ones. To better measure the scalability of tools, we split each of these benchmarks into several new instances, each instance testing a single parameter value. Code duplication between various instances of the same benchmark was avoided by means of the `#include` directive introduced in the REC-2017 language.
– When introducing parameterized instances, we eliminated a few parameter values that were so large that no tool could feasibly handle them.
– Conversely, some parameter values were too small, and we tried to increase them, still making sure that one tool at least could handle them.
– Similarly, certain benchmarks (such as `langton`*) did non-trivial calculations, but were still too easy for most tools. We made these benchmarks more complex by iterating their calculations over a domain of values.

In addition to this work on existing benchmarks, we introduced new benchmarks that had never been used for the Rewrite Engines Competitions:

– We added two instances `tak18` and `tak38` of the Takeuchi function, which was one of the Kranz-Stansifer benchmarks [42] not already included in the set of REC benchmarks.
– We added a new benchmark `intnat` that defines signed integers and their related operations according to the axiomatization $\mathbf{F}_2^2$ proposed in [15]. This benchmark contains 1900+ tests to check that multiply, divide, and modulo operations are correctly defined.
– We added eight new benchmarks (`add`*, `mul`*, and `omul`*) that define binary adders and multipliers operating on 8-bit, 16-bit, and 32-bit machine words. Each of these benchmarks contains 4000+ tests to make sure that results are correctly computed.
– We added a large benchmark `maa`, which features a cryptographic algorithm formerly used to authenticate financial transactions [19]; this algorithm is described as a large term rewrite system (13 sorts, 18 constructors, 644 non-constructors, and 684 rewrite rules). The corresponding REC-2017 specification is 1575-line long; in comparison, the largest benchmarks inherited from the three former Rewrite Engines Competitions have less than 300 lines.

Doing so, we obtained a collection[11] of 85 benchmarks totalling more than 40,000 lines of code written in the REC-2017 language. We organized this collection into two parts: 15 "simple" benchmarks, which (almost) all the tools listed in Sect. 2 can handle successfully, and 70 "non-trivial" benchmarks, some of which are likely to be significantly more challenging for the tools.

6 Language Conciseness

As a by-product of our study, we can compute, for each language L, a *verbosity estimation* metric. This metric is not based on the number of code lines, a

[11] http://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS.

traditional measure that is much too subjective, as it depends on how often the translator targeting language L inserts line breaks.

Instead, our metric is defined as follows: we concatenate the source files of all the 85 benchmarks encoded in language L, and count the number of lexical tokens (i.e., keywords, identifiers, punctuations, mathematical symbols, etc.) present in these files. Compound symbols, such as "`->`", "`==`", or "`()`", count only for one.

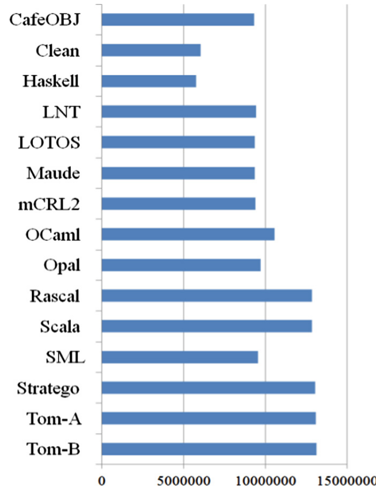
This metric based on the number of tokens is quite objective, as it quantifies how many symbols a human should insert to get a working program. Moreover, counting tokens avoids the subjective debate of keywords, e.g., "**begin**" and "**end**" versus "{" and "}". Such quantitative feedback can be useful to language developers: if a language L' requires twice as many tokens as another language L to encode the same problem, then language L' is likely to be more difficult to learn and use by humans.

Table 6 summarizes our measurements. One can thus classify the languages in four categories: the "concise" ones (Clean and Haskell), the "normal" ones (CafeOBJ, LNT, LOTOS, Maude, mCRL2, Opal, and SML), a "slightly verbose" one (OCaml), and the "verbose" ones (Rascal, Scala, Stratego, and Tom).

Such results deserve a few comments: (i) The well-known conciseness of Clean and Haskell is confirmed; (ii) OCaml and SML suffer from the lack of predefined printers for values of constructor types, so that a printing function must be explicitly defined for each type; (iii) Opal lacks predefined printers too, but also lacks predefined structural equality to compare values of constructor types; (iv) Rascal, Scala, Stratego/XT, and Tom have large numbers of tokens partly because every call to a operation with arity zero must, according to Java conventions, be followed by a "`()`" token; notice that such large numbers probably

Table 6. Verbosity estimation of all languages, measured on the 85 REC benchmarks

language	# tokens
CafeOBJ[a]	9,294,297
Clean	6,037,299
Haskell	5,754,474
LNT	9,395,563
LOTOS	9,334,063
Maude	9,313,456
mCRL2	9,354,987
OCaml	10,565,289
Opal	9,701,085
Rascal	12,845,312
REC-2017	9,173,664
Scala	12,825,710
SML	9,517,116
Stratego	13,020,691
Tom-A	13,065,023
Tom-B	13,106,915



[a]CafeOBJ-A and CafeOBJ-B have the same number of tokens

arise from constructors with arity zero, since SML, which requires "`()`" only after non-constructors with arity zero, has a much lower number of tokens.

7 Execution Platform

To assess the tools listed in Sect. 2 in a reproducible manner, we installed them on two separate machines, each working with a single user, local file systems only (no NAS, NFS, Samba, etc.), and in stand-alone mode (no remote administration by computer staff, no automatic download of patches, etc.). Such constraints forced us to reuse retired servers.

As for the processors, we selected the widespread x86 (32-bit) and x64 (64-bit) architectures. We used a Sun Ultra 20 M2 server (2007) powered by one AMD Opteron 1210 (x86, dual core, 1.8 GHz) with 2 GB RAM, and a Transtec 2500 L server (2004) powered by two AMD Opteron 246 (x64, single core, 2.0 GHz) with 16 GB RAM.

As for the operating system, we selected Linux because it is commonplace in software competitions and because several of the tools listed on Sect. 2 are not available on Windows. We chose the stable Debian release (Debian Linux 8 "Jessie"). For the Java-based tools (Rascal, Scala, and Tom) we installed OpenJDK (version 1.8.0-91).

Because some REC-2017 benchmarks manipulate large algebraic terms and certain tools make heavy use of recursion, the maximal stack size had to be increased; we set it to 32 MB on x86 and 512 MB on x64. For the Java-based tools, we set the JVM stack size to the same value and increased the overall JVM memory size to the maximum.

Because some tools seem to run forever or, at least, take much longer than others (we sometimes observed differences in two orders of magnitude), we used the Linux `timeout` command to allocate each tool a maximum amount of time. We set the timeout value to 360 s (i.e., six minutes) at most per benchmark — the choice of this value is justified hereafter. For the few tools that protect themselves against interrupts by catching signals, the uncatchable POSIX signal SIGKILL is used to force termination.

To collect execution statistics, we use the `memtime` utility originally developed by the Uppaal team in 2002 and later enhanced at INRIA Grenoble[12]. Among the six values returned by `memtime` after each execution, we only retain the exit status (zero if ok, non-zero otherwise) and wall-clock time. The four remaining values (CPU time, memory usage, etc.) are not relevant in our context. For instance, memory usage is not meaningful, as it only concerns the main process: if a tool is provided as a shell script that launches child processes in sequence or in parallel, then `memtime` will only report the memory consumption of that shell script itself, ignoring all its child processes.

Thus, tools are launched as follows: "`memtime timeout 360` *tool options...*" and their execution can terminate in four different ways: *success* (normal completion, exit code is zero), *failure* (failed execution, exit code is non-zero, meaning

[12] http://cadp.inria.fr/resources/#memtime (version 1.4).

that the tool properly halted, declaring it cannot tackle the problem), *crash* (abnormal interrupt by a signal, usually SIGSEV or SIGBUS, meaning that the tool terminated abruptly, often due to a programming error, such as dereferencing a null pointer, or to a stack overflow not handled properly), or *timeout* (interrupt when the wall-clock time exceeds the specified duration).

Because tools can fail, crash, or time out on different benchmarks, determining a performance ranking between tools is a non-trivial problem, since dilemmas arise from conflicting criteria. For instance, how to compare a tool that computes for a long time until it is halted by timeout with another tool that quickly stops and declares that it cannot solve the problem? Both tools have failed, and the former has even taken more time than the latter! Such issues have been studied, e.g., in [23] in the particular case of planning algorithms for robotics and artificial intelligence. "*Any comparison, competitions especially, has the unenviable task of determining how to trade-off or combine the three metrics (number solved, time, and number of steps).*" Based on this remark, we adopted two complementary metrics:

– Our first metric is the *score*, which counts how many benchmarks have been successfully tackled by a given tool within the specified timeout duration. This is the standard solution advocated in [23]: "*Because no planner has been shown to solve all possible problems, the basic metric for performance is the number or percentage of problems actually solved within the allowed time. This metric is commonly reported in the competitions.*"
– Our second metric is the *user time*, defined as the total wall-clock time spent by a given tool on a given benchmark, from the moment a tool is invoked until the tool terminates or is halted. The user time is always less or equal to the specified timeout duration. It is a somewhat heterogeneous metric that covers all steps used in problem solving, and not only the time spent in "pure" rewriting. For an interpreter, the user time measures the time spent in processing the benchmark. For a compiler, the user time is the sum of the time spent in compiling the benchmark source file to binary code and the time spent in executing this binary code. The user time also includes the time needed to parse input files, the time taken by the C compiler for tools that generate C code, the time taken by the Lisp interpreter for tools that generate Lisp code, the time taken to launch, warm up, and halt the JVM for tools generating Java code, etc.

To determine a performance ranking between tools, we combine these two metrics (score, user time) using a lexicographic order, considering score first, and then user time only to distinguish between tools having the same score.

We developed a highly automated execution platform, with two families of shell scripts: the `run*` scripts to launch tools on benchmarks and collect execution results, and the `tab*` scripts to analyze these data and build spreadsheet files containing global statistics in CSV format.

8 Experimental Results

Table 7 summarizes the results obtained when running all the tools on the 70 "non-trivial" REC-2017 benchmarks, with a timeout set to 360 s. In each table cell of the form "X / Y", both values X and Y are sums computed over all the 70 benchmarks; X refers to the x86 platform (2 GB RAM)[13] and Y to the x64 platform (16 GB RAM)[14]. The last two columns of Table 7 give our two chosen metrics: score and user time.

Many useful findings can be drawn from Table 7. First, the score values should not be understood as absolute numbers; especially, the 50% score value should not be seen as a threshold separating "good" tools from "bad" ones. Indeed, if

Table 7. Execution results for the 70 REC benchmarks on 32-bit/64-bit platforms

Tool	Successes	Failures	Crashes	Timeouts	Score	Time (seconds)
CafeOBJ-A	31/31	8/4	0/4	31/31	44.3%/44.3%	12490/13946
CafeOBJ-B	38/38	15/8	0/3	17/21	54.3%/54.3%	8561/10170
Clean	30/40	30/22	8/1	2/7	42.9%/57.1%	805/2632
Clean hack	54/54	10/10	0/0	6/6	77.1%/77.1%	2623/2697
Haskell	70/68	0/0	0/0	0/2	100%/97.1%	1867/2091
LNT	62/63	8/7	0/0	0/0	88.6%/90.0%	1135/3028
LOTOS	62/63	8/7	0/0	0/0	88.6%/90.0%	990/2884
Maude	64/67	2/0	1/0	3/3	91.4%/95.7%	2095/2122
mCRL2 jitty	44/46	0/0	6/2	20/22	62.9%/65.7%	9586/9914
mCRL2 jittyc	48/52	0/0	4/0	18/18	68.6%/74.3%	8148/8101
OCaml compiler	64/64	0/0	0/0	6/6	91.4%/91.4%	2772/2718
OCaml interpreter	60/61	2/2	0/0	8/7	85.7%/87.1%	3937/3855
Opal	57/59	1/0	2/1	10/10	81.4%/84.3%	4759/4817
Rascal compiler	34/37	4/2	0/0	32/31	48.6%/52.9%	14486/14286
Rascal interpreter	37/40	2/0	0/0	31/30	52.9%/57.1%	12274/12322
Scala	49/49	10/7	0/0	11/14	70.0%/70.0%	6092/7147
SML: MLton	58/58	5/5	0/0	7/7	82.9%/82.9%	4622/4997
SML: SML/NJ	52/52	5/5	0/0	13/13	74.3%/74.3%	5672/5628
Stratego	48/49	3/0	0/0	19/21	68.6%/70.0%	8260/9062
Tom-A	51/52	3/1	0/0	16/17	72.9%/74.3%	7578/8064
Tom-B	60/60	3/1	0/0	7/9	85.7%/85.7%	5438/5887

[13] The detailed 32-bit results are available from https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/results-rec/2018-04-07-overview-360-32.csv?view=log and https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/results-rec/raw-v2/2018-04-07-rec360-32.csv?view=log.

[14] The detailed 64-bit results are available from https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/results-rec/2018-04-05-overview-360-64.csv?view=log and https://gforge.inria.fr/scm/viewvc.php/rec/2015-CONVECS/results-rec/raw-v2/2018-04-05-rec360-64.csv?view=log.

the 15 "simple" benchmarks mentioned in Sect. 5 were also taken into account in Table 7, then all scores would be above 52%, as each tool can tackle each simple benchmark in two minutes at most.

Also, it appears that most of the 70 benchmarks are "difficult". Precisely, only 21% (on 32 bits) and 27% (on 64 bits) of these benchmarks can be successfully tackled by all the tools within six minutes, meaning that all the other benchmarks make at least one tool fail, crash, or time out.

It is expected from a software competition to rank tools according to their performance and merits. As discussed above, we rank the tools according to a lexicographic ordering that first compares the scores and, if the scores are identical, compares the total time spent. For conciseness, we only discuss the tools present on the Top-5 podium, i.e., those 5 (or 7) tools (out of 21) that can solve at least 85% of the 70 REC benchmarks, using at most 360 s per benchmark:

1. **Haskell ranks first** and, in particular, performs clearly better than any other functional language. This result seems in contradiction with prior studies, e.g., [25], which concluded that Clean was faster than Haskell, and the Computer Language Benchmarks Game[15], which reports that OCaml is often faster than Haskell. This could be explained by improvements brought to the GHC compiler over the last ten years (i.e., since the publication of [25]) and by the fact that our study strictly focuses on constructor types and pattern matching, whereas many examples of the Computer Language Benchmarks Game deal with numbers or arrays and rely on programmer's skills, which may strongly affect performance results for a particular tool.
2. **Maude ranks second**, very close to Haskell on the 64-bit platform. This good score is remarkable for at least three reasons: (i) Maude works as an interpreter, not a compiler; (ii) its interpreter does not seem to exploit the "`[ctor]`" annotation used to distinguish constructors from non-constructors; (iii) although Maude has many sophisticated features (subsorts, associativity/commutativity/identity properties, reflection, strategies, modules, object-orientation, etc.), it does not neglect to do basic things (i.e., plain rewriting) efficiently.
3. **OCaml ranks third**. More precisely, the compiled version of OCaml (based on the native-code generator "`ocamlopt.opt`") takes the 3rd position, while the interpreted version (based on the bytecode generator "`ocamlc.opt`" and the bytecode interpreter "`ocamlrun`") takes the 6th position.
4. **CADP ranks fourth**. More precisely, its LOTOS compiler CÆSAR.ADT takes the 4th position and its LNT translator LNT2LOTOS takes the 5th position, reflecting the fact that LNT is first translated to LOTOS before being compiled using CÆSAR.ADT. The implementations of these languages are fast, but sometimes fail due to memory exhaustion, which might be solved by enabling the garbage collector and/or the hash-consing optimization, none of which is activated by default. Also, the present study contradicts the results of [44] claiming that mCRL2 performs better than LOTOS — notice that

[15] http://benchmarksgame.alioth.debian.org/u64q/ocaml.html.

another study on the benchmarking of model checking tools [32] further disconfirms the claims of [44].

5. **Tom ranks fifth**, as the Tom-B translation variant takes the 7th position, with a score of 85.7%.

Interestingly, the Top 5 podium is the same on 32-bit and 64-bit platforms, although the numerical score of each tool might slightly differ on both platforms.

One may also question the influence of timeouts — see the related discussion (*Are Time Cut-offs Unfair?*) in [23, pp. 18 and 28]. Our timeout value was set to 360 s, so that the full cycle of experiments on all benchmarks takes roughly 24 h, thus allowing daily progress on the study. Retrospectively, this value is appropriate, as it does not prevent the best tool (i.e., Haskell) from reaching a score of 100%, meaning that each of our benchmarks can feasibly be solved within 360 s.

Short timeout values give a significant advantage to tools that compute fast; larger timeout values might deeply modify the scores displayed in the Table 7 if the failure numbers of the "timeouts" column are transferred to the "successes" column. To investigate this possibility, we increased our timeout value to 1800 s (i.e., at most 30 min per benchmark) and redid our experiments, which took many days this time. Such a larger timeout value especially benefits Rascal, CafeOBJ, Scala, and Stratego/XT, whose scores increase by (up to) +18.6%, +11.6%, +7.1%, and +5.7% respectively. However, the Top 5 podium remains unchanged. It is worth noticing that on the 64-bit platform, Maude now reaches a 100% score, like Haskell, but still remains in second position, as it appears to be 30% slower than Haskell on average.

9 Threats to Validity

Our results might be affected by various limitations, which we discuss hereafter:

- *Missing languages/tools.* Even though our study takes into account a much larger number of languages and tools than the former Rewrite Engines Competitions, we might have, due to lack of time and knowledge, omitted some prominent languages and tools. This does not affect the validity of our results, but calls for new, broader studies in the future.
- *Missing benchmarks.* We took great care to have a collection of benchmarks as large and as diverse as possible, both by reusing classical benchmarks from prior competitions and by developing new original benchmarks. Similar collections of term rewrite systems perhaps exist, although we did not find them by searching the Web nor by asking senior researchers in the field. As they are, the results of our study provide nevertheless valuable information that tool developers can exploit without the need for additional benchmarks.
- *Platform specifics.* The execution platform used for this study perhaps lacks diversity: the 32-bit and 64-bit machines both use old AMD Opteron processors and the same operating system. Conducting experiments with other processors and operating systems would be worthwhile; yet, x86/x64 and Linux

are massively present in software competitions and research labs. Also, the Java-based tools have only been assessed using OpenJDK; their performance might be slightly improved using Oracle JDK, the proprietary implementation of Java.

- *Undetected errors.* Our execution platform does not check whether tool results (i.e., the ground terms obtained after evaluating the terms listed in each `EVAL` section) are correct, because each tool displays these results in a custom format. We checked most results visually, but not automatically.
- *Imprecise measurements.* In our study, each tool is considered as a "black box", from the point of view of an ordinary user more interested in tool performance than in internal algorithmics. Obtaining finer information, such as the time specifically spent in "pure" rewriting, and properly estimating the resources consumed by third-party software (C compiler, Lisp interpreter, Java runtime, etc.) would require the active participation of all tool developers to make such data available.
  Also, enforcing limitations on resource usage and accurately measuring the time and memory consumed by software tools is a difficult task [2]. In this respect, the `memtime` and `timeout` commands only deliver approximate results; in particular, wall-clock time does not distinguish between sequential and multithreaded executions (although our outdated machines partly avoid this issue, as both can run at most two processes or two threads in parallel). More precise measurements of memory footprint and multi-core usage could be obtained using the BenchKit [27] or BenchExec [2] platforms intended for software competitions.
- *Misbehaving tools.* In our experiments, all tools are run in sequence on all benchmarks. There is always a possibility that a tool misbehaves and perturbs, either inadvertently or on purpose, the subsequent runs of other tools on the benchmarks, e.g., by leaving huge files that decrease available disk size or swap space, by forking processes that survive after the tool's termination and steal computing resources, or even by modifying the user's `crontab` file to launch periodic tasks. In our study, such issues are quite unlikely because experiments were conducted with already existing tools, but software competitions in general should address security issues and properly isolate tools using, e.g., virtual machines, as in BenchKit [27], or the `cgroups` feature of the Linux kernel, as in BenchExec [2].
- *Translation biases.* We observed that the main factor influencing the score of certain tools was (together with the amount of memory allocated to Java virtual machines) the quality of the translation from REC-2017 to the input languages of these tools. Depending on the translation, results can indeed greatly vary, as can be seen with Tom-A and Tom-B. We believe that our REC translators behave fairly: (i) they do not implement ad hoc optimizations to favour certain tools; (ii) they have not been designed to work against particular tools either: when performance issues were observed, translation outputs were sent to tool developers, whose feedback often helped to improve the translators; (iii) the generated files are available online for cross inspection by all stakeholders. In this respect, our fully-automated approach

significantly improves over the former Rewrite Engines Competitions, in which some translations were done manually, thus allowing custom optimizations to be introduced for specific tools.
Actually, we made two minor exceptions to this neutrality principle: (i) because the Opal compiler does not tolerate very long lines, our translator to Opal inserts a newline every twenty commas; this solved problems for three benchmarks; (ii) because the Java Virtual Machine has a limitation that methods cannot be larger than 64 kbytes, our translators to Rascal, Scala, and Tom (all based on Java) split each large `main` method into several smaller methods containing no more than 90 or 100 instructions each.
– *Misinterpretation of results.* Our study would not have been possible without the impressive work done by language designers and tool developers. Keeping this in mind, the results of Sect. 8 should not be construed as an absolute ranking of languages and tools, for at least two reasons: (i) our study focuses on certain specific language features, and (ii) our two metrics, score and user time, encompass diverse algorithmic activities, many of which are performed by third-party software (C compiler, Lisp interpreter, Java runtime, etc.).

10 Conclusion

The present study builds upon the Rewrite Engines Competitions (2006, 2008, and 2010), but significantly extends them by bringing more languages/tools, more benchmarks, and full automation. These three former competitions led to somewhat inconclusive results, namely that tools were difficult to compare because they had different application domains, different strengths, and different weaknesses. Our study avoids this drawback by focusing on the most widely used fragment of term rewriting, keeping only those language features understood by all the tools (i.e., types defined by free constructors, conditional term rewrite systems that are deterministic and terminating, and evaluation of closed terms), excluding advanced features (e.g., predefined types, array types, higher-order functions, rewriting modulo associativity/commutativity/identity, context-sensitive rewriting, strategies, etc.) that are not implemented or not identically implemented across all the tools. Within these restrictions, our study highlights a large body of common traits shared by algebraic, functional, and object-oriented languages.

There have been recent efforts to define, for a large class of conditional term rewrite systems, a notion of complexity that approximates the maximal number of rewrite steps performed when reducing a term to its normal form [26]. Our compared evaluation of actual implementations can be seen as a practical complement to these theoretical results, possibly with a triple impact:

– Our experimental results refute the common beliefs that “speed performance is not an issue” and that “all tools are more or less linear in performance”. The global success score measured on all 70 benchmarks varies between 50% and 100% across tools. The time required to process a large benchmark such as the MAA [19] varies by more than two orders of magnitude between the

fastest tool and the slowest tool. Clearly, not all implementations are alike: performance and scalability definitely matter. In some cases (see Sect. 8), our results contradict conclusions from earlier studies or, at least, bring complementary information.
- Numerous scientific articles have been published about term rewriting engines and pattern-matching compiling algorithms, but it is unclear which ones are the most efficient in practice. Our software platform provides a basis for undertaking such a study. Should a systematic comparison be too demanding, it might be sufficient to learn from the tools (e.g., Haskell or Maude) that score high on the REC-2017 benchmarks.
- Our results may provide an incentive for tool developers to reconsider, and possibly improve, the way in which constructor types, term rewriting, and/or pattern-matching are implemented in their tools. Radical decisions could even be taken, for instance, generating code in a higher-level language (e.g., Haskell) than C or Java, in order to defer the implementation of constructor types and pattern matching to a compiler shown to handle them efficiently.

Besides our evaluation results, our main contribution is a software platform[16] for benchmarking implementations of term rewriting and pattern matching. This platform has five components: (i) the REC-2017 language, which is a revised version of REC-2008 [12, Sect. 3] [11, Sect. 3.1]; (ii) a set of translators that automatically convert REC-2017 into 17 different languages; (iii) a collection of 85 benchmarks, consisting of (deterministic, free-constructor, left-linear, many-sorted, and terminating) conditional term rewrite systems expressed in the REC-2017 language, including closed terms to be evaluated; (iv) the output files generated by applying the translators to these benchmarks, thus providing test cases of possible interest to tool developers; and (v) a set of scripts for automatically running all the tools on all the benchmarks, recording execution results, and computing statistics.

This software platform already served in two recent case studies: a specification of the MAA cryptographic function [19] and an elegant definition of signed integers using term rewrite systems [15]. It could also be the starting point for re-launching the Rewrite Engines Competitions, which would progress the state of the art and address some of the limitations mentioned in Sect. 9.

Acknowledgements. We are grateful to Marc Brockschmidt (Cambridge/AProVE), Francisco Durán (Malaga/Maude), Steven Eker (Stanford/Maude), Florian Frohn (Aachen/AProVE), Carsten Fuhs (London/AProVE), John van Groningen (Nijmegen/Clean), Jan Friso Groote (Eindhoven/mCRL2), Paul Klint (Amsterdam/Rascal), Pieter Koopman (Nijmegen/Clean), Davy Langman (Amsterdam/Rascal), Xavier Leroy (Paris/OCaml), Florian Lorenzen (Berlin/Opal), Pierre-Etienne Moreau (Nancy/Tom), Jeff Smits (Delft/Stratego), Jurriën Stutterheim (Nijmegen/Clean), and

[16] It is available on the SVN server of INRIA and can be obtained using the command "`svn checkout svn://scm.gforge.inria.fr/svnroot/rec/2015-CONVECS`", or "`svn checkout svn://scm.gforge.inria.fr/svnroot/rec`" to obtain also all the benchmarks used in the three former REC competitions.

Eelco Visser (Delft/Stratego) for their patient explanations and help concerning their respective tools. The present paper also benefited from discussions with Bertrand Jeannet (Grenoble), Fabrice Kordon (Paris), and Jose Meseguer (Urbana-Champaign).

References

1. Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., Reilles, A.: Tom: piggybacking rewriting on Java. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 36–47. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73449-9_5
2. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement – Application to automatic verification. In: Fischer, B., Geldenhuys, J. (eds.) SPIN 2015. LNCS, vol. 9232, pp. 160–178. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_12
3. van den Bos, J., Hills, M., Klint, P., van der Storm, T., Vinju, J.J.: Rascal: from algebraic specification to meta-programming. In: Durán, F., Rusu, V. (eds.) Proceedings of the 2nd International Workshop on Algebraic Methods in Model-based Software Engineering (AMMSE 2011), Zurich, Switzerland. Electronic Proceedings in Theoretical Computer Science, vol. 56, pp. 15–32, June 2011
4. van den Brand, M., Klint, P., Olivier, P.: Compilation and memory management for ASF+SDF. In: Jähnichen, S. (ed.) CC 1999. LNCS, vol. 1575, pp. 198–213. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-540-49051-7_14
5. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17 - a language and toolset for program transformation. Sci. Comput. Program. **72**(1–2), 52–70 (2008)
6. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LNT to LOTOS Translator (Version 6.6), INRIA, Grenoble, France, February 2017
7. Clavel, M., et al.: Maude Manual (Version 2.7.1), July 2016
8. Denker, G., Talcott, C.L., Rosu, G., van den Brand, M., Eker, S., Serbanuta, T.: Rewriting logic systems. Electronic Notes Theor. Comput. Sci. **176**(4), 233–247 (2007)
9. 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)
10. Didrich, K., Fett, A., Gerke, C., Grieskamp, W., Pepper, P.: OPAL: Design and implementation of an algebraic programming language. In: Gutknecht, J. (ed.) Programming Languages and System Architectures. LNCS, vol. 782, pp. 228–244. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-57840-4_34
11. Durán, F., et al.: The third rewrite engines competition. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 243–261. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16310-4_16
12. Durán, F., et al.: The second rewrite engines competition. Electron. Notes Theor. Comput. Sci. **238**(3), 281–291 (2009)
13. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1 - Equations and Initial Semantics, EATCS Monographs on Theoretical Computer Science, vol. 6. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-69962-7
14. Garavel, H.: Compilation of LOTOS Abstract Data Types. In: Vuong, S.T. (ed.) Proceedings of the 2nd International Conference on Formal Description Techniques FORTE 1989, Vancouver B.C., Canada, pp. 147–162. North-Holland, December 1989

15. Garavel, H.: On the most suitable axiomatization of signed integers. In: James, P., Roggenbach, M. (eds.) WADT 2016. LNCS, vol. 10644, pp. 120–134. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72044-9_9
16. Garavel, H., Lang, F., Mateescu, R.: Compiler construction using LOTOS NT. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 9–13. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_3
17. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. Int. J. Softw. Tools Technol. Transfer (STTT) **15**(2), 89–107 (2013)
18. Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: Katoen, J.-P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd. LNCS, vol. 10500, pp. 3–26. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68270-9_1
19. Garavel, H., Marsso, L.: A large term rewrite system modelling a pioneering cryptographic algorithm. In: Hermanns, H., Höfner, P. (eds.) Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems (MARS 2017), Uppsala, Sweden. Electronic Proceedings in Theoretical Computer Science, vol. 244, pp. 129–183, April 2017
20. Garavel, H., Turlier, P.: CÆSAR.ADT : un compilateur pour les types abstraits algébriques du langage LOTOS. In: Dssouli, R., von Bochmann, G. (eds.) Actes du Colloque Francophone pour l'Ingénierie des Protocoles (CFIP 1993), Montréal, Canada. pp. 325–339. Hermès, Paris, September 1993
21. Giesl, J., et al.: Proving termination of programs automatically with AProVE. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 184–191. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_13
22. Groote, J., Mousavi, M.: Modeling and Analysis of Communicating Systems. The MIT Press, Cambridge (2014)
23. Howe, A.E., Dahlman, E.: A critical assessment of benchmark comparison in planning. J. Artif. Intell. Res. **17**, 1–33 (2002)
24. ISO/IEC: LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization - Information Processing Systems - Open Systems Interconnection, Geneva, September 1989
25. Jansen, J.M., Koopman, P., Plasmeijer, R.: From interpretation to compilation. In: Horváth, Z., Plasmeijer, R., Soós, A., Zsók, V. (eds.) CEFP 2007. LNCS, vol. 5161, pp. 286–301. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88059-2_8
26. Kop, C., Middeldorp, A., Sternagel, T.: Complexity of Conditional Term Rewriting. Logical Methods Comput. Sci. **13**(1) (2017)
27. Kordon, F., Hulin-Hubard, F.: BenchKit, a tool for massive concurrent benchmarking. In: Proceedings of the 4th International Conference on Application of Concurrency to System Design (ACSD 2014), Tunis La Marsa, Tunisia, pp. 159–165. IEEE Computer Society, June 2014
28. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml System Release 4.04 - Documentation and User's Manual. INRIA, Paris, France, March 2016
29. Leung, A.: Nowhere: A Pattern Matching Tool for Standard ML - Version 1.1 Manual. New York University, NY (2000)
30. Marlow, S. (ed.): Haskell 2010 Language Report, April 2010

31. Mateescu, R., Garavel, H.: XTL: a meta-language and tool for temporal logic model-checking. In: Margaria, T. (ed.) Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT 1998), Aalborg, Denmark, pp. 33–42. BRICS, July 1998
32. Mazzanti, F., Ferrari, A.: Ten diverse formal models for a CBTC automatic train supervision system. In: Gallagher, J.P., van Glabbeek, R., Serwe, W. (eds.) Proceedings of the 3rd Workshop on Models for Formal Analysis of Real Systems and the 6th International Workshop on Verification and Program Transformation (MARS/VPT 2018), Thessaloniki, Greece. Electronic Proceedings in Theoretical Computer Science, vol. 268, pp. 104–149, April 2018
33. Middeldorp, A., Hamoen, E.: Completeness results for basic narrowing. Appl. Algebra Eng. Commun. Comput. **5**, 213–253 (1994)
34. Milner, R., Tofte, M., Harper, R., MacQueen, D.: Definition of Standard ML (Revised). MIT Press, May 1997
35. Odersky, M., et al.: The Scala Language Specification - Version 2.11. Programming Methods Laboratory. EPFL, Switzerland, March 2016
36. Pepper, P., Lorenzen, F. (eds.): The Programming Language Opal - 6th Corrected Edition. Department of Software Engineering and Theoretical Computer Science, Technische Universität Berlin, Germany, October 2012
37. Plasmeijer, R., van Eekelen, M., van Groningen, J.: Clean Version 2.2 Language Report. Department of Software Technology, University of Nijmegen, The Netherlands, December 2011
38. van de Pol, J.: Just-in-time: on strategy annotations. Electron. Notes Theor. Comput. Sci. **57**, 41–63 (2001)
39. Pol, J.: JITty: a rewriter with strategy annotations. In: Tison, S. (ed.) RTA 2002. LNCS, vol. 2378, pp. 367–370. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45610-4_26
40. Ponsini, O., Fédèle, C., Kounalis, E.: Rewriting of imperative programs into logical equations. Sci. Comput. Program. **56**(3), 363–401 (2005)
41. Schnoebelen, P.: Refined compilation of pattern-matching for functional languages. Sci. Comput. Program. **11**, 133–159 (1988)
42. Stansifer, R.: Imperative versus functional. SIGPLAN Not. **25**(4), 69–72 (1990)
43. Thompson, S.J.: Laws in Miranda. In: Scherlis, W.L., Williams, J.H., Gabriel, R.P. (eds.) Proceedings of the ACM Conference on LISP and Functional Programming (LFP 1986), Cambridge, Massachusetts, USA, pp. 1–12 (1986)
44. van Weerdenburg, M.: An account of implementing applicative term rewriting. Electron. Notes Theor. Comput. Sci. **174**(10), 139–155 (2007)

Multi-paradigm Programming in Maude

Santiago Escobar(✉)

DSIC-ELP, Universitat Politècnica de València, Valencia, Spain
sescobar@dsic.upv.es

Abstract. Modern multi-paradigm programming languages combine the most important features of functional programming, logic programming, concurrent programming, and constraint programming. Multi-paradigm programming applied to the Maude specification language would replace the functional viewpoint by an equational viewpoint while retaining and extending the other features. A former paper illustrated how many features available in modern functional logic languages are easily definable and simulated in Maude, and showed how Maude goes beyond standard practices in the functional logic area by using, e.g. equational properties such as associativity and commutativity or order-sorted information. As a practical application the paper used the *Missionaries and Cannibals* equational logic program given by Goguen and Meseguer for Eqlog in the eighties. However, no relevant execution results were included due to the preliminary implementation of narrowing in Maude. In this paper we provide more relevant execution results by changing the focus from narrowing with rules (i.e., symbolic reachability) into narrowing with variant equations (i.e., equational unification), the latter now implemented in Maude 2.7.1 with very good performance.

1 Introduction

Functional logic programming [23–25] combines the most important features of functional programming such as Haskell and logic programming such as Prolog. From the functional paradigm it borrows algebraic data types, advanced typing, evaluation strategies, and high-order functions among other features; and from the logic paradigm it borrows logical variables, computing with partial information, constraint solving, and nondeterministic search for solutions. Functional logic programming in the Maude specification language combines logical variables, computing with partial information, and constraint solving with reasoning modulo equational properties, advanced data types, order-sorted typing, efficient equational evaluation, distinction between concurrent and functional parts, and parameterised modules.

This work was partially supported by the EU (FEDER) and the Spanish MINECO under grant TIN 2015-69175-C4-1-R, by the Spanish Generalitat Valenciana under grant PROMETEOII/2015/013, and by the US Air Force Office of Scientific Research under award number FA9550-17-1-0286.

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 26–44, 2018.
https://doi.org/10.1007/978-3-319-99840-4_2

Modern multi-paradigm programming languages [26] currently involve different paradigms in a seamless way: (i) functional programming, (ii) logic programming, (iii) concurrent programming, and (iv) constraint programming. Modern multi-paradigm programming languages use an evaluation mechanism called *narrowing*. Narrowing is a generalization of term rewriting that allows free variables in terms (as in logic programming) and replaces pattern matching by unification in order to (non-deterministically) reduce these terms. Narrowing was originally introduced for automated theorem proving [42], then used as a mechanism for solving equational unification problems [19], it became the "de facto" evaluation mechanism for functional logic programming languages [4], and it was generalized from equational unification problems to solve the more general problem of symbolic reachability [38] and, in a more modern perspective, of *logical* model checking in [6,17]. The narrowing mechanism has a number of important applications including automated proofs of termination [5], execution of functional logic programming languages [4], program transformation [1], verification of cryptographic protocols [38], equational unification [28], and SMT [37,45], just to mention just a few.

The Eqlog programming language [21] developed by Goguen and Meseguer in the eighties was a first attempt to combine both equational programming with logic programming. Eqlog unified equational programming and Horn-logic programming into one paradigm. Its logic design task was to embed order-sorted equational logic and Horn logic without equality into a suitable Horn logic with equality [22]. Since the eighties, Jose Meseguer has been interested in including logical features into Maude (see his paper [34] dedicated to Goguen's 65th birthday) but most of the appropriate technology was missing. Many researchers in the functional logic programming area (see [11,20,23,32,40]) have also tried, since the eighties, to combine the best features of both paradigms and many possibilities have been explored (see [23,24] for a survey). Nowadays there is a remarkable body of programming languages and tools in the functional logic area. Many people strongly believe that Maude with logical features (as a modern successor of Eqlog) would be an excellent choice in the near future for multi-paradigm programming: (i) equational programming, (ii) object-oriented programming, (iii) concurrent programming, (iv) logic programming, and (v) constraint solving. For instance, Jose Meseguer recently recalled many of the symbolic capabilities currently available in Maude in [36].

Order-sorted unification and narrowing modulo axioms became first available in 2009 as part of the Maude 2.4 release [8]. Unification became available as a built-in feature in Maude while narrowing was available in Full Maude, an extension of Maude written in Maude itself. Unification worked for any combination of symbols being either free or associative-commutative (AC). Narrowing worked for modules having only rules and axioms and relied on the built-in unification algorithm. It supported the concept of symbolic reachability analysis of terms with logical variables, computing suitable substitutions for the variables in both the origin and the destination terms [17].

Unification and narrowing were updated in 2011 as part of the Maude 2.6 release [14]. First, built-in unification was extended to allow any combination of symbols being either free, commutative (C), associative-commutative (AC), or associative-commutative with an identity symbol (ACU). The performance was dramatically improved, allowing further development of other techniques in Maude. Second, the concept of *variant* [10] was added to Maude. The introduction of variants led to a significant improvement in Maude's reasoning capabilities: variant generation, variant-based unification, and symbolic reachability based on variant-based unification became all available for the first time. However, all the variant-based features and the narrowing-based reachability were only available in Full Maude, and for a restricted class of theories called *strongly right irreducible*.

Unification and narrowing were extended again as part of the Maude 2.7 release [13]. First, the built-in unification algorithm allows any combination of symbols being free, C, AC, ACU, CU (commutativity and identity), U (identity), Ul (left identity), and Ur (right identity). Second, variant generation and variant-based unification are implemented as built-in features in Maude. This built-in implementation works for any convergent theory modulo the axioms described above, both allowing very general equational theories (beyond the strongly right irreducible ones) and boosting the performance not only of these features but of their applications. Third, narrowing-based reachability is still only available in Full Maude, but uses the built-in variant-based unification.

Recently, unification and narrowing were extended once more as part of the Maude 2.7.1 release [12]. The last addition has been *associative unification*. The range of the following additional symbolic reasoning features were substantially extended with reasoning modulo associativity: (i) variant generation; (ii) variant unification; and (iii) narrowing-based symbolic reachability analysis.

In [16] I showed how many modern programming features in programming languages such as Curry [25] can be implemented using Maude 2.6. I recalled the existing implementation of narrowing as symbolic reachability in Maude 2.6, illustrated how many features available in Curry are easily definable and simulated in Maude, and showed how Maude goes beyond standard practices in the functional logic area by using, e.g. equational properties such as associativity and commutativity or order-sorted information. As a practical application the paper used the *Missionaries and Cannibals* equational logic program given by Goguen and Meseguer for Eqlog in the eighties. However, no relevant execution results were included due to the preliminary implementation of narrowing in Maude 2.6. In this paper I provide more relevant execution results by changing the focus from narrowing with rules (i.e., symbolic reachability) into narrowing with variant equations (i.e., equational unification), the latter now implemented in Maude 2.7.1 with very good performance. Furthermore, since Maude now supports associative unification, the *Missionaries and Cannibals* specification has been updated.

In Sect. 2 we present the *Missionaries and Cannibals* equational logic program of [21]. In Sect. 3 we introduce some basic concepts on rewriting logic.

In Sect. 4 we recall the built-in variant-based equational unification and how it is made available in Maude. In Sect. 5 we show how the *Missionaries and Cannibals* equational logic program of [21] can be written using the narrowing features described in [16]. In Sect. 6 we demonstrate how queries on the motivating example are effectively executed using built-in variant-based equational unification. Finally, we conclude in Sect. 7.

2 Example: Missionaries and Cannibals

As a motivating example for the reader, we present the *Missionaries and Cannibals* equational logic program of [21]. The equational logic program[1] of [21] used a syntax proposed for Eqlog where a functional syntax very close to Maude was combined with some syntax for Horn-clauses using the symbol ":-".

```
module MAC[T :: MACTH] using NAT, TRIPLIST = LIST[trip] is
  preds
    boatok : trip
    solve, good : triplist
  fns
    boat : pred -> trip
    lb,rb : triplist -> pset
    mset,cset : pset -> pset
  vars
    PS:pset, L:triplist, P:person, T:trip
  axioms
    boatok(boat(PS)) :- # PS = 1.
    boatok(boat(PS)) :- # PS = 2.
    mset(PS) = PS /\ m0.
    cset(PS) = PS /\ c0.
    lb(nil) = m0 + c0.
    rb(nil) = empty.
    lb(L * boat(PS)) = lb(L) - PS :- even # L.
    rb(L * boat(PS)) = rb(L) + PS :- even # L.
    rb(L * boat(PS)) = rbQ - PS :- odd # L.
    lb(L * boat(PS)) = lb(L) + PS :- odd # L.
    good(L * T)
      :- # cset(lb(L * T)) =< # mset(lb(L * T)) or mset(lb(L * T)) = 0,
         # cset(rb(L * T)) =< # mset(rb(L * T)) or mset(rb(L * T)) = 0,
         good(L), boatok(T).
    good(nil).
    solve(L) :- good(L), lb(L) = empty.
endmod MAC
```

[1] The original program in [21] had an error because the number of cannibals has to be lower than or equal to the number of missionaries in both sides unless there is no missionary. We discovered this error thanks to the new automated reasoning capabilities in Maude.

This module is parametric on a theory `T : : MACTH` for the names of the missionaries and cannibals, which are instantiated to `m0 = taylor, helen, william` and `c0 = umugu, nzwave, amoc`. Also a module for lists is imported, where `_*_` is the constructor symbol for lists and `#_` is the length operation for lists. The system is configured as a list of trips (sort `triplist`) where each trip is a term rooted by a predicate `boat` with a set of names of missionaries and cannibals. Each trip in the list is considered `good` if it satisfies some properties. Odd positions in the list represent moving from left to right and even positions from right to left. There are some extra symbols for set manipulation: `_+_` for union, `_-_` for removal, and `_/_` for intersection. Note that lists and (multi-)sets are the data structures in this example with associativity, commutativity and/or identity axioms. There are also some symbols for lists: `even` indicates whether a list has an even number of elements and `odd` indicates whether a list has an odd number of elements. Finally, the predicate `boatok` checks whether a trip is ok and `solve` is the general predicate for checking/generating the `triplist` solution.

This is an example requiring some constraint solving features, logical variables, order-sorted types, associativity for lists, and associativity, commutativity, and identity for multisets. For instance, it requires constraint solving features because of the numerical conditions for length of lists in the conditions of predicate `good`; this would be solved by using a generator function and using these length functions by residuation. Also, the program considers equational properties, since the problem is represented by a list of trips and each element is the boat with a multiset of missionaries and cannibals; this would be easily handled by narrowing modulo these properties. And it clearly includes order-sorted information in the sense of having people which are specialized into missionaries and cannibals.

Note that some inherently functional logic features [23–25] are necessary here: (i) a semantics of values instead of all reachable terms, (ii) predicates are just conditional rules evaluated into a special sort, different from `Bool`, that only contains positive (or successful) cases, and (iii) conditions in conditional rules are indeed strict equalities instead of syntactic equality.

3 Background on Rewriting Logic and Term Rewriting

We follow the classical notation and terminology from [43] for term rewriting, and from [31] for rewriting logic and order-sorted notions. We assume an order-sorted signature $\mathbf{\Sigma} = (\mathsf{S}, \leq, \Sigma)$ with poset of sorts $(\mathsf{S}, \leq)$ and such that for each sort $\mathsf{s} \in \mathsf{S}$ the connected component of s in $(\mathsf{S}, \leq)$ has a top sort, denoted $[\mathsf{s}]$, and all $f : \mathsf{s_1} \cdots \mathsf{s_n} \rightarrow \mathsf{s}$ with $n \geq 1$ have a top sort overloading $f : [\mathsf{s_1}] \cdots [\mathsf{s_n}] \rightarrow [\mathsf{s}]$. We also assume an S-sorted family $\mathcal{X} = \{\mathcal{X}_\mathsf{s}\}_{\mathsf{s} \in \mathsf{S}}$ of disjoint variable sets with each $\mathcal{X}_\mathsf{s}$ countably infinite. $\mathcal{T}_{\Sigma}(\mathcal{X})_\mathsf{s}$ is the set of terms of sort s, and $\mathcal{T}_{\Sigma,\mathsf{s}}$ is the set of ground terms of sort s. We write $\mathcal{T}_{\Sigma}(\mathcal{X})$ and $\mathcal{T}_{\Sigma}$ for the corresponding order-sorted term algebras. For a term t, $Var(t)$ denotes the set of variables in t.

Positions are represented by sequences of natural numbers denoting an access path in the term when viewed as a tree. The top or root position is denoted by the empty sequence Λ. We define the relation $p \leq q$ between positions as $p \leq p$ for any p; and $p \leq p.q$ for any p and q. Given $U \subseteq \Sigma \cup \mathcal{X}$, $Pos_U(t)$ denotes the set of positions of a term t that are rooted by symbols or variables in U. The set of positions of a term t is written $Pos(t)$, and the set of non-variable positions $Pos_\Sigma(t)$. The subterm of t at position p is $t|_p$ and $t[u]_p$ is the term t where $t|_p$ is replaced by u.

A *substitution* $\sigma \in \mathcal{S}ubst(\Sigma, \mathcal{X})$ is a sorted mapping from a finite subset of $\mathcal{X}$ to $\mathcal{T}_\Sigma(\mathcal{X})$. Substitutions are written as $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ where the domain of σ is $Dom(\sigma) = \{X_1, \ldots, X_n\}$ and the set of variables introduced by terms $t_1, \ldots, t_n$ is written $Ran(\sigma)$. The identity substitution is *id*. Substitutions are homomorphically extended to $\mathcal{T}_\Sigma(\mathcal{X})$. The application of a substitution σ to a term t is denoted by $t\sigma$. For simplicity, we assume that every substitution is idempotent, i.e., σ satisfies $Dom(\sigma) \cap Ran(\sigma) = \emptyset$. Substitution idempotency ensures $t\sigma = (t\sigma)\sigma$. The restriction of σ to a set of variables V is $\sigma|_V$; sometimes we write $\sigma|_{t_1,\ldots,t_n}$ to denote $\sigma|_V$ where $V = Var(t_1) \cup \cdots \cup Var(t_n)$. Composition of two substitutions σ and σ' is denoted by $\sigma\sigma'$.

A *Σ-equation* is an unoriented pair $t = t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}}$ for some sort $\mathsf{s} \in \mathsf{S}$. Given an order-sorted signature Σ and a set $\mathcal{E}$ of Σ-equations, order-sorted equational logic induces a congruence relation $=_\mathcal{E}$ on terms $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$ (see [33]). The $\mathcal{E}$-equivalence class of a term t is denoted by $[t]_\mathcal{E}$ and $\mathcal{T}_{\Sigma/\mathcal{E}}(\mathcal{X})$ and $\mathcal{T}_{\Sigma/\mathcal{E}}$ denote the corresponding order-sorted term algebras modulo $\mathcal{E}$. Throughout this paper we assume that $\mathcal{T}_{\Sigma,\mathsf{s}} \neq \emptyset$ for every sort s, because this affords a simpler deduction system.

An *equational theory* $(\Sigma, \mathcal{E})$ is a pair with Σ an order-sorted signature and $\mathcal{E}$ a set of Σ-equations. The *$\mathcal{E}$-subsumption* preorder $\sqsupseteq_\mathcal{E}$ (or just $\sqsupseteq$ if $\mathcal{E}$ is understood) holds between $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$, denoted $t \sqsupseteq_\mathcal{E} t'$ (meaning that t is *more general* than t' modulo $\mathcal{E}$), if there is a substitution σ such that $t\sigma =_\mathcal{E} t'$; such a substitution σ is said to be an *$\mathcal{E}$-match* from t to t'.

An *$\mathcal{E}$-unifier* for a Σ-equation $t = t'$ is a substitution σ such that $t\sigma =_\mathcal{E} t'\sigma$. For $Var(t) \cup Var(t') \subseteq W$, a set of substitutions $CSU^W_\mathcal{E}(t = t')$ is said to be a *complete* set of unifiers for the equality $t = t'$ modulo $\mathcal{E}$ away from W iff: (i) each $\sigma \in CSU^W_\mathcal{E}(t = t')$ is an $\mathcal{E}$-unifier of $t = t'$; (ii) for any $\mathcal{E}$-unifier ρ of $t = t'$ there is a $\sigma \in CSU^W_\mathcal{E}(t = t')$ such that $\sigma|_W \sqsupseteq_\mathcal{E} \rho|_W$; (iii) for all $\sigma \in CSU^W_\mathcal{E}(t = t')$, $Dom(\sigma) \subseteq (Var(t) \cup Var(t'))$ and $Ran(\sigma) \cap W = \emptyset$. If the set of variables W is irrelevant or is understood from the context, we write $CSU_\mathcal{E}(t = t')$ instead of $CSU^W_\mathcal{E}(t = t')$. An $\mathcal{E}$-unification algorithm is *complete* if for any equation $t = t'$ it generates a complete set of $\mathcal{E}$-unifiers. A unification algorithm is said to be *finitary* and complete if it always terminates after generating a finite and complete set of solutions.

A *rewrite rule* is an oriented pair $l \to r$, where[2] $l \notin \mathcal{X}$ and $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ for some sort $\mathsf{s} \in \mathsf{S}$. An *(unconditional) order-sorted rewrite theory* is a triple $(\Sigma, \mathcal{E}, R)$ with Σ an order-sorted signature, $\mathcal{E}$ a set of Σ-equations, and R a set of rewrite rules.

The rewriting relation on $\mathcal{T}_\Sigma(\mathcal{X})$, written $t \to_R t'$ or $t \to_{p,R} t'$ holds between t and t' iff there exist $p \in Pos_\Sigma(t)$, $l \to r \in R$ and a substitution σ, such that $t|_p = l\sigma$, and $t' = t[r\sigma]_p$. The relation $\to_{R/\mathcal{E}}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $=_\mathcal{E}; \to_R; =_\mathcal{E}$, i.e., $t \to_{R/\mathcal{E}} t'$ iff there exists u, u' s.t. $t =_\mathcal{E} u \to_R u' =_\mathcal{E} t'$. Note that $\to_{R/\mathcal{E}}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ induces a relation $\to_{R/\mathcal{E}}$ on the free $(\Sigma, \mathcal{E})$-algebra $\mathcal{T}_{\Sigma/\mathcal{E}}(\mathcal{X})$ by $[t]_\mathcal{E} \to_{R/\mathcal{E}} [t']_\mathcal{E}$ iff $t \to_{R/\mathcal{E}} t'$. The transitive (resp. transitive and reflexive) closure of $\to_{R/\mathcal{E}}$ is denoted $\to^+_{R/\mathcal{E}}$ (resp. $\to^*_{R/\mathcal{E}}$).

The application of one $\to_{R/\mathcal{E}}$ step is undecidable in general since $\mathcal{E}$-congruence classes can be arbitrarily large. Therefore, $R/\mathcal{E}$-rewriting is usually implemented [29] by R,$\mathcal{E}$-rewriting. A relation $\to_{R,\mathcal{E}}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is defined as: $t \to_{p,R,\mathcal{E}} t'$ (or just $t \to_{R,\mathcal{E}} t'$) iff there exist $p \in Pos_\Sigma(t)$, a rule $l \to r$ in R, and a substitution σ such that $t|_p =_\mathcal{E} l\sigma$ and $t' = t[r\sigma]_p$.

We assume that the relation $\to_{R,\mathcal{E}}$ is locally *$\mathcal{E}$-coherent* [29], i.e., $\forall t_1, t_2, t_3$ we have $t_1 \to_{R,\mathcal{E}} t_2$ and $t_1 =_\mathcal{E} t_3$ implies $\exists t_4, t_5$ such that $t_2 \to^*_{R,\mathcal{E}} t_4$, $t_3 \to^+_{R,\mathcal{E}} t_5$, and $t_4 =_\mathcal{E} t_5$. Let us recall how coherence works at least for the common associative-commutative (AC) case. The best way to illustrate it is by its *absence*. Consider a symbol _+_ declared as AC. Now consider the rule $b + b \to c$, where b and c are constants. This rule, if not completed by another, is *not* coherent modulo AC. What this means is that there will be term *contexts* in which the rule *should* be applied, but it cannot be applied. Consider, for example, the term $b + (a + b)$, where a is also a constant. Intuitively, we should be able to apply to it the above rule to simplify it to the term $a + c$ in one step. However, since we are using the weaker rewrite relation $\to_{R,AC}$ instead of the stronger but much harder to implement relation $\to_{R/AC}$, we cannot! The problem is that the rule cannot be applied (even if we match modulo AC) to either the top term $b + (a + b)$ or the subterm $a + b$. We can however make our rule *coherent* modulo AC by adding the extra rule $b + b + Y \to c + Y$. This extended version of our rule will now apply to the term $b + (a + b)$, giving the simplification $b + (a + b) \longrightarrow_{R,AC} a + c$. Technically, what coherence means is that the weaker relation $\to_{R,\mathcal{E}}$ becomes semantically equivalent to the stronger relation $\to_{R/\mathcal{E}}$.

Coherence can be handled implicitly or explicitly, i.e., either the matching mechanism is modified to take care of this issue or the rules are explicitly extended, which is the option shown above; see [46] for a comparison between implicit and explicit extensions. For rewriting, implicit extensions are sufficient in many cases, as the implicit $\mathcal{E}$-coherence completion provided by the Maude

[2] Note that we do not impose here the standard condition $Var(r) \subseteq Var(l)$, necessary for executability of rewriting in practice. Rewriting with extra variables in right-hand sides is handled at a theoretical level by allowing the matching substitution to instantiate these extra variables in any possible way. Extra variables do no pose any problem to narrowing and are part of the nondeterministic search of solutions typical of logic programming.

tool [9] for any combination of associativity (A), commutativity (C), and identity (U) axioms. For narrowing, implicit extension is more complicated and it is sufficient to consider explicit single-variable extensions in common cases such as combinations of C, AC, and ACU axioms, i.e., given a rule $s \to t$ one considers $s + x \to t + x$ where x is a new variable. The method is as follows for AC. For any symbol f which is AC, and for any rule of the form $f(u, v) \to w$ in $\mathcal{E}$, we add also the equation $f(f(u, v), X) \to f(w, X)$, where X is a new variable not appearing in u, v, w. In an order-sorted setting, we should give to X *the biggest sort possible*, so that it will apply in all generality. As an additional optimization, note that some rules may already be coherent modulo AC, so that we need not add the extra equation. See [15,35] for further information.

We also assume that the equational theory is split into $\mathcal{E} = E \cup Ax$ such that E is a set of equations oriented into rules and Ax is a set of equational axioms satisfying:

1. Ax is *regular*, i.e., for each $t = t'$ in Ax, we have $Var(t) = Var(t')$, and *sort-preserving*, i.e., for each substitution σ, we have $t\sigma \in \mathcal{T}_{\Sigma}(\mathcal{X})_{\mathsf{s}}$ iff $t'\sigma \in \mathcal{T}_{\Sigma}(\mathcal{X})_{\mathsf{s}}$; furthermore, for each equation $t = t'$ in Ax, all variables in $Var(t)$ have a top sort.
2. Ax has a finitary and complete unification algorithm, which implies that Ax-matching is finitary and complete.
3. For each $t \to t'$ in E we have $Var(t') \subseteq Var(t)$.
4. E is *sort-decreasing*, i.e., for each $t \to t'$ in E, each $\mathsf{s} \in \mathsf{S}$, and each substitution σ, $t'\sigma \in \mathcal{T}_{\Sigma}(\mathcal{X})_{\mathsf{s}}$ implies $t\sigma \in \mathcal{T}_{\Sigma}(\mathcal{X})_{\mathsf{s}}$.
5. The relation $\to_{E,Ax}$ is confluent, terminating, and locally Ax-*coherent*, i.e., for each term t, the relation terminates and produces a unique irreducible term (up to Ax-equivalence) denoted by $t{\downarrow}_{E,Ax}$.

Given an order-sorted equational theory $(\Sigma, E \cup Ax)$, (t', θ) is an E, Ax-*variant* [18] (or just a variant) of term t if $t\theta{\downarrow}_{E,Ax} =_{Ax} t'$ and $\theta{\downarrow}_{E,Ax} =_{Ax} \theta$. Given two term variants $(t_1, \theta_1), (t_2, \theta_2)$ of a term t, we write $(t_1, \theta_1) \sqsupseteq_{E,Ax} (t_2, \theta_2)$, meaning (t_1, θ_1) is a more general variant of t than (t_2, θ_2), iff there is a substitution ρ such that $(\theta_1\rho)|_{Var(t)} =_{Ax} \theta_2|_{Var(t)}$ and $t_1\rho =_{Ax} t_2{\downarrow}_{E,Ax}$. An order-sorted equational theory $(\Sigma, E \cup Ax)$ has the *finite variant property* [18] iff for each Σ-term t, a complete set of its most general variants is finite. A finitary and complete unification algorithm is defined for order-sorted equational theories with the finite variant property [18].

4 Narrowing in Maude

Logic programming languages are well suited for goal solving. Functional programming languages are equipped with equational definition of operations. Several approaches have been considered in the literature for combining the functional and logic paradigms, see [23]. On the one hand, it is a natural idea to add an equality predicate to logic programs, leading to equational logic programming [27]. On the other hand, it is also a natural idea to add logical variables

to functional programs, leading to narrowing-based equational reasoning [19]. Logic variables are also valuable at the level of model checking rather than functional programming, as proposed for symbolic reachability in [38] and extended to *logical* model checking in [6,17].

At each rewriting step one must choose which subterm of the subject term and which rule of the specification are going to be considered. Similarly, at each narrowing step one must choose which subterm of the subject term, which rule of the specification, and which instantiation[3] on the variables of the subject term and the rule's left-hand side are going to be considered. The difference between a rewriting step and a narrowing step is that, in both cases we use a rewrite rule $l \to r$ to rewrite t at a position p in t, but narrowing unifies the left-hand side l and the chosen subject term $t|_p$ before actually performing the rewriting step. Narrowing is restricted[4] to non-variable positions of t, whereas rewriting does not require such a restriction.

Let $\mathcal{R} = (\Sigma, E \cup Ax, R)$ be an order-sorted rewrite theory where R is a set of unconditional rewrite rules, specified with the `rl` keyword, E is a set of unconditional equations specified with the `eq` and `variant` keywords, and Ax is a set of commonly occurring axioms—declared in Maude as equational attributes—such that an $E \cup Ax$-unification procedure is available in Maude. Unification algorithms already available in Maude are divided in two groups: (i) Ax-unification for order-sorted signatures with any combination of free, C, AC, or ACU function symbols [8], and (ii) $E \cup Ax$-unification for order-sorted equational theories with the finite variant property [14].

Let $CSU_{E\cup Ax}(u = u')$ provide[5] a finitary and complete set of unifiers for any pair of terms u, u' with the same top sort. The R,$(E \cup Ax)$-*narrowing* relation on $\mathcal{T}_\Sigma(\mathcal{X})$ is defined as $t \rightsquigarrow_{\sigma,p,R,E\cup Ax} t'$ (or $\rightsquigarrow_\sigma$ when p, R, E, Ax are understood) if there is a non-variable position $p \in Pos_\Sigma(t)$, a (possibly renamed) rule $l \to r$ in R, and a unifier $\sigma \in CSU_{E\cup Ax}(t|_p = l)$ such that $t' = (t[r]_p)\sigma$. We denote by $t \rightsquigarrow^+_{\sigma,R,E\cup Ax} t'$ (resp. $t \rightsquigarrow^*_{\sigma,R,E\cup Ax} t'$) the transitive (resp. reflexive-transitive) closure of the narrowing relation, where σ is obtained as the composition of the substitutions for each narrowing step in the sequence.

Consider the following system module defining the addition function `_+_` on natural numbers built from `0` and `s`:

```
mod NAT-NARROWING is
  sort Nat .
  op 0 : -> Nat [ctor] .
```

[3] Demand-driven narrowing strategies may require instantiations of a term that do not correspond to a most general unifier of a subterm and a left-hand side, see [2,3].

[4] The *paramodulation* inference rule used in paramodulation-based theorem proving [39] is similar to narrowing and does not require non-variable positions.

[5] In the present implementation of Maude, we are not interested in a minimal set of unifiers, but only in a finite and complete set. Minimality is easily achieved in syntactic unification (see [30]) but it is very costly in Ax-unification or $E \cup Ax$-unification, e.g., the ACU-unification available in Maude does not always provide a minimal set of unifiers.

```
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> [Nat] .
  vars X Y : Nat .
  rl [base] : 0 + Y => Y .
  rl [ind] : s(X) + Y => s(X + Y) .
endm
```

Consider the term `X + s(0)` and the two rules `base` and `ind`. Narrowing will instantiate[6] variable `X` with `0` and `s(X')`, respectively. The following two narrowing steps are generated:

$$\begin{array}{l} \mathtt{X} + \mathtt{s}(0) \leadsto_{\{\mathtt{X}\mapsto 0\},\mathtt{base}} \mathtt{s}(0) \\ \mathtt{X} + \mathtt{s}(0) \leadsto_{\{\mathtt{X}\mapsto \mathtt{s}(\#1:\mathtt{Nat})\},\mathtt{ind}} \mathtt{s}(\#1:\mathtt{Nat} + \mathtt{s}(0)) \end{array}$$

Note that, for simplicity, we show only the bindings of the unifier that affect the input term. There are infinitely many narrowing derivations starting at the input expression `X + s(0)` (at each step the reduced subterm is underlined):

1. $\underline{\mathtt{X} + \mathtt{s}(0)} \leadsto_{\{\mathtt{X}\mapsto 0\},\mathtt{base}} \mathtt{s}(0)$
2. $\underline{\mathtt{X} + \mathtt{s}(0)} \leadsto_{\{\mathtt{X}\mapsto \mathtt{s}(\#1:\mathtt{Nat})\},\mathtt{ind}} \mathtt{s}(\underline{\#1:\mathtt{Nat} + \mathtt{s}(0)}) \leadsto_{\{\#1:\mathtt{Nat}\mapsto 0\},\mathtt{base}} \mathtt{s}(\mathtt{s}(0))$
3. $\underline{\mathtt{X} + \mathtt{s}(0)} \leadsto_{\{\mathtt{X}\mapsto \mathtt{s}(\#1:\mathtt{Nat})\},\mathtt{ind}} \mathtt{s}(\underline{\#1:\mathtt{Nat} + \mathtt{s}(0)})$
   $\leadsto_{\{\#1:\mathtt{Nat}\mapsto \mathtt{s}(\#2:\mathtt{Nat})\},\mathtt{ind}} \mathtt{s}(\mathtt{s}(\underline{\#2:\mathtt{Nat} + \mathtt{s}(0)})) \leadsto_{\{\#2:\mathtt{Nat}\mapsto 0\},\mathtt{base}} \mathtt{s}(\mathtt{s}(\mathtt{s}(0)))$

And some of those infinitely many narrowing derivations are infinite in length, e.g. by applying rule `ind` infinitely many times:

$$\begin{array}{l} \underline{\mathtt{X} + \mathtt{s}(0)} \leadsto_{\{\mathtt{X}\mapsto \mathtt{s}(\#1:\mathtt{Nat})\},\mathtt{ind}} \mathtt{s}(\underline{\#1:\mathtt{Nat} + \mathtt{s}(0)}) \\ \quad \leadsto_{\{\#1:\mathtt{Nat}\mapsto \mathtt{s}(\#2:\mathtt{Nat})\},\mathtt{ind}} \mathtt{s}(\mathtt{s}(\underline{\#2:\mathtt{Nat} + \mathtt{s}(0)})) \\ \quad \leadsto_{\{\#2:\mathtt{Nat}\mapsto \mathtt{s}(\#3:\mathtt{Nat})\},\mathtt{ind}} \mathtt{s}(\mathtt{s}(\mathtt{s}(\underline{\#3:\mathtt{Nat} + \mathtt{s}(0)}))) \\ \quad \ldots \end{array}$$

The narrowing relation currently available in Maude is slightly different than the standard one formally defined above. Let $\mathcal{R} = (\Sigma, G \cup E \cup Ax, R)$ be an order-sorted rewrite theory where R, E, and Ax are defined as above and G are the remaining equations. Note that equations in G do not have the `variant` attribute and have no restriction, i.e., they can be conditional equations, with the `owise` attribute, etc. Each narrowing step of the form $t \leadsto_{\sigma,p,R,E\cup Ax} t'$ is followed by simplification $t'{\downarrow_{G,Ax}}$, i.e., the combined relation is defined as $t \leadsto_{\sigma,p,R,E,G,Ax} t''$ iff $t \leadsto_{\sigma,p,R,E\cup Ax} t'$ and $t'' = t'{\downarrow_{G,Ax}}$. Note that this combined relation may be incomplete because equations G are not considered for unification, i.e., given a reachability problem of the form $\exists \overline{x} : t(\overline{x}) \to^* t'(\overline{x})$ and a solution σ (i.e., $t\sigma \to^*_{R,G\cup E\cup Ax} t'\sigma$), the relation $\leadsto_{\sigma,p,R,E,G,Ax}$ may not be able to find a more general solution.

[6] New variables in Maude are introduced as `#1:Nat` or `%1:Nat` instead of `X'`.

4.1 The Variant Narrowing Strategy

Applying narrowing without any restriction is very wasteful when you have an equational theory $(\Sigma, E \cup Ax)$. First, for computing variants in a decomposition we are only interested in normalized substitutions, so we can restrict our interest to narrowing derivations that provide only normalized substitutions. Our second idea is to give priority to rewrite steps w.r.t. unrestricted narrowing steps. Thanks to convergence modulo Ax, as soon as a general rewrite (or narrowing) step is enabled in a term that has also narrowing steps such a more general step is always taken before any further narrowing steps are applied.

For example, consider a different version of the NAT module using variant equations that includes an extra redundant equation:

```
fmod NAT-VARIANT-1 is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> [Nat] .
  vars X Y Z : Nat .
  eq [base] : 0 + Y = Y [variant] .
  eq [ind] : s(X) + Y = s(X + Y) [variant] .
  eq [red] : X + s(0) = s(X) [variant] .
endfm
```

As explained before, there are infinitely many narrowing derivations starting at the input expression X + s(0) but there is a most general narrowing derivation: $\underline{\mathtt{X} + \mathtt{s}(\mathtt{0})} \leadsto_{id,\mathtt{red}} \mathtt{s}(\mathtt{X})$. Variant narrowing will discard all the other narrowing steps and keep just the most general one. See [18] for details.

4.2 The Folding Variant Narrowing Strategy

The variant narrowing strategy defined above is a strategy in the sense that always returns a subset of the narrowing steps available for each term. Note that it has no memory of previous steps, just the input term to be narrowed, hence it incurs no memory overhead. However, more sophisticated strategies can be developed by introducing some sort of memory that can avoid the repeated generation of useless or unnecessary computation steps. This is the case of the folding narrowing strategy of [18], which, when combined with the variant narrowing strategy, provides the *folding variant narrowing strategy* which is complete for variant generation of a term and it terminates when the term has a finite set of *most general variants.*

For example, consider a different version of the NAT module using variant equations that includes several extra equations:

```
fmod NAT-VARIANT-2 is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
```

```
  op _+_ : Nat Nat -> [Nat] .
  vars X Y Z : Nat .
  eq [base-left] : 0 + Y = Y [variant] .
  eq [ind-left] : s(X) + Y = s(X + Y) [variant] .
  eq [base-right] : X + 0 = X [variant] .
  eq [ind-right] : X + s(Y) = s(X + Y) [variant] .
endfm
```

Given the input expression X + s(0) there is a most general folding variant narrowing derivation: $\underline{\texttt{X} + \texttt{s(0)}} \leadsto_{id,\texttt{ind-right}} s(\underline{\texttt{X} + \texttt{0}}) \leadsto_{id,\texttt{base-right}} \texttt{s(X)}$. Folding variant narrowing will discard all the other narrowing derivations and keep just the most general one. Note that the input expression X + Y still gives infinitely many narrowing derivations and some of them are infinite in length.

4.3 Variant-Based Unification in Maude 2.7.1

The most recent Maude 2.7.1 version [7] provides a command for $E \cup Ax$-equational unification based on the folding variant narrowing strategy above. Consider the NAT-VARIANT-2 module above, where one can obtain an incremental number of unifiers for a given unification problem. On the one hand, it is possible to have a finite number of most general unifiers for a unification problem although the theory does not have the finite variant property. For instance, the unification problem between X + s(0) and s(s(s(0))) returns just one unifier.

```
Maude> variant unify in NAT-VARIANT-2 : X + s(0) =? s(s(s(0))) .

Unifier #1
X --> s(s(0))
```

On the other hand, we can approximate the number of unifiers of a unification problem that we suspect does not have a finite number of most general unifiers. For instance, the unification problem between terms X + Y and Z has many solutions and variant-based unification does not stop, since folding variant narrowing keeps generating variants for X + Y. However, we can obtain some solutions by including a bound in the command.

```
Maude> variant unify [4] in NAT-VARIANT-2 : X + Y =? Z  .

Unifier #1        Unifier #2       Unifier #3        Unifier #4
X --> 0           X --> #1:Nat     X --> s(0)        X --> %1:Nat
Y --> #1:Nat      Y --> 0          Y --> %1:Nat      Y --> s(0)
Z --> #1:Nat      Z --> #1:Nat     Z --> s(%1:Nat)   Z --> s(%1:Nat)
```

5 Missionaries and Cannibals Using Symbolic Reachability

In [16], we gave a specification of the original equational logic program by using features and functionalities of Maude.

```
mod MAC is
 pr SUCCESS . pr TRIPLIST . pr PSET .

 ops taylor helen william : -> Elem [ctor] .
 ops umugu nzwawe amoc : -> Elem [ctor] .
 var L : TripList . var T : Trip . var PS : PSet .

 op gen : Elem -> [Success] .
 rl gen(taylor) => success .            eq gen(taylor) = success .
 rl gen(helen) => success .             eq gen(helen) = success .
 rl gen(william) => success .           eq gen(william) = success .
 rl gen(umugu) => success .             eq gen(umugu) = success .
 rl gen(nzwawe) => success .            eq gen(nzwawe) = success .
 rl gen(amoc) => success .              eq gen(amoc) = success .

 op m0 : -> [PSet] .                    op c0 : -> [PSet] .
 eq m0 = taylor helen william  .        eq c0 = umugu nzwawe amoc  .

 op mset : PSet -> [PSet] .             op cset : PSet -> [PSet] .
 eq mset(PS) = PS /\ m0 .               eq cset(PS) = PS /\ c0 .

 op boatok : Trip -> [Success] .        op boat : PSet -> Trip [ctor] .
 eq boatok(boat(X:Elem)) = gen(X:Elem) .
 eq boatok(boat(X1:Elem X2:Elem))
  = gen(X1:Elem) >> gen(X2:Elem) >> ((X1:Elem =/= X2:Elem) =:= true) .

 ops lb rb : TripList -> [PSet] .
 eq lb(nil) = m0 c0 .
 eq lb(L * boat(PS))= if (even # L) then (lb(L) - PS) else (lb(L) PS) fi .
 eq rb(nil) = empty .
 eq rb(L * boat(PS)) = if (even # L) then (rb(L)PS) else (rb(L) - PS) fi .

 op good : TripList -> [Success] .
 eq good(nil) = success .
 eq good(L * T)
  =  boatok(T) >> good(L) >>
       ( (# cset(lb(L * T)) =< # mset(lb(L * T))
          or (# mset(lb(L * T)) == 0)  )
         and
         (# cset(rb(L * T)) =< # mset(rb(L * T))
          or (# mset(rb(L * T)) == 0))           ) =:= true .

 op solve : TripList -> [Success] .
 eq solve(L) = good(L) >> (lb(L) == empty) =:= true .
endm
```

We used kinds for defined operations and regular sorts for constructor symbols, following the semantics of values. One of the features of functional logic programming languages called *residuation* is obtained by including rules and equations

for most operations, i.e., terms are simplified using the equations before any rule is tried by narrowing.

We used the sort Success for predicates and the constant success as a remanent of true, avoiding problems of negation in logic programming.

```
sort Success .
op success : -> Success [ctor] .
```

We used the symbol >> for conditional evaluation of constraints, such that the left side is evaluated before the right side. Note that this order of evaluation requires the use of frozenness arguments and evaluation strategies.

```
op _>>_ : [Success] [Success] -> [Success]
                                [frozen (2) strat (1 0) gather (e E)] .
rl success >> X:[Success] => X:[Success] .
eq success >> X:[Success] = X:[Success] .
```

Functional logic programming languages use strict equality instead of syntactic equality and we have defined the symbol =:= for strict equality. The only sort used for strict equality in the example is Bool, and we defined =:= for Bool only in the positive case returning success:

```
op _=:=_ : Bool Bool -> [Success] [comm] .
rl X:Bool =:= X:Bool => success .
eq X:Bool =:= X:Bool = success .
```

And again as described in [16], functional logic programming languages usually allow the user to have some control on constraint solving capabilities thanks to the use of >> and also data generators, so that the search is more controlled and follows a generate-and-test approach. We use a specific generator function gen. Operators gen, >> and =:= are the only ones defined using both rules and equations. Finally, logic predicates are defined on the sort Success and are considered as conditional equations but using the >> operator instead of being standard conditional equations in Maude.

In [16], the auxiliary module TRIPLIST for list of trips was defined using just a head-tail approach for lists (no associativity axiom) whereas the auxiliary module PSET for sets of missionaries and cannibals were defined using associativity, commutativity, and identity axioms.

The experiments were performed in [16] using the narrowing-based search command available in Full Maude 2.6, but were very preliminary and could considered only problems with very few variables. For example, having two variables E:Elem and E1:Elem for two places in a specific solution to the problem:

```
search  solve(nil * boat(taylor umugu) * boat(taylor) * boat(nzwawe amoc) * boat(umugu) *
                    boat(william helen) * boat(helen nzwawe) * boat(taylor helen) *
                    boat(E1) * boat(umuguE1) * boat(E) * boat(E nzwawe) ) ~>! success .
Solution 1                               Solution 2
E1:Elem --> amoc ; E:Elem --> amoc      E1:Elem --> amoc ; E:Elem --> helen
Solution 3                               Solution 4
E1:Elem --> amoc ; E:Elem --> taylor    E1:Elem --> amoc ; E:Elem --> umugu
Solution 5                               Solution 6
E1:Elem --> amoc ; E:Elem --> william   E1:Elem --> nzwawe ; E:Elem --> amoc
Solution 7
E1:Elem --> nzwawe ; E:Elem --> umugu
```

6 Missionaries and Cannibals Using Variant Equations

The new specification using variant equations is similar to the previous section but we have replaced all rules by equations labeled with the `variant` flag.

```
op gen : Elem -> [Success] .
eq gen(taylor) = success [variant] .
eq gen(helen) = success [variant] .
eq gen(william) = success [variant] .
eq gen(umugu) = success [variant] .
eq gen(nzwawe) = success [variant] .
eq gen(amoc) = success [variant] .
```

We have also replaced the rule for symbol `_=:=_` by a variant equation. That is, for each sort, the strict equality is always defined only in the positive case returning `success`, e.g. for the sort `Nat` is defined as follows:

```
op _=:=_ : Nat Nat -> [Success] [comm] .
eq X:Bool =:= X:Bool = success [variant] .
```

Since Maude 2.7.1 allows associative unification, we added associativity to the auxiliary module for list of trips.

```
sorts TripList Trip .
subsort Trip < TripList .
op nil : -> TripList [ctor] .
op _*_ : TripList TripList -> TripList [ctor assoc] .
```

The new experiments simply used the variant-based equational unification to find all thirty six solutions to the problem.

```
Maude> variant unify solve(nil * boat(E1:Elem E2:Elem) * boat(E1:Elem) * boat(E3:Elem E4:Elem)
* boat(E2:Elem) * boat(E5:Elem E6:Elem) * boat(E6:Elem E3:Elem) * boat(E1:Elem E6:Elem)
* boat(E4:Elem) * boat(E2:Elem E4:Elem) * boat(E6:Elem) * boat(E6:Elem E3:Elem)) =? success  .
```

```
Unifier #1
E1:Elem --> helen E2:Elem --> amoc E3:Elem --> umugu E4:Elem --> nzwawe E5:Elem --> william E6:Elem --> taylor
Unifier #2
E1:Elem --> william E2:Elem --> amoc E3:Elem --> umugu E4:Elem --> nzwawe E5:Elem --> helen E6:Elem --> taylor
Unifier #3
E1:Elem --> helen E2:Elem --> nzwawe E3:Elem --> umugu E4:Elem --> amoc E5:Elem --> william E6:Elem --> taylor
Unifier #4
E1:Elem --> william E2:Elem --> nzwawe E3:Elem --> umugu E4:Elem --> amoc E5:Elem --> helen E6:Elem --> taylor
Unifier #5
E1:Elem --> taylor E2:Elem --> amoc E3:Elem --> umugu E4:Elem --> nzwawe E5:Elem --> william E6:Elem --> helen
Unifier #6
E1:Elem --> william E2:Elem --> amoc E3:Elem --> umugu E4:Elem --> nzwawe E5:Elem --> taylor E6:Elem --> helen
Unifier #7
E1:Elem --> taylor E2:Elem --> nzwawe E3:Elem --> umugu E4:Elem --> amoc E5:Elem --> william E6:Elem --> helen
Unifier #8
E1:Elem --> william E2:Elem --> nzwawe E3:Elem --> umugu E4:Elem --> amoc E5:Elem --> taylor E6:Elem --> helen
Unifier #9
E1:Elem --> taylor E2:Elem --> amoc E3:Elem --> umugu E4:Elem --> nzwawe E5:Elem --> helen E6:Elem --> william
Unifier #10
E1:Elem --> helen E2:Elem --> amoc E3:Elem --> umugu E4:Elem --> nzwawe E5:Elem --> taylor E6:Elem --> william
Unifier #11
E1:Elem --> taylor E2:Elem --> nzwawe E3:Elem --> umugu E4:Elem --> amoc E5:Elem --> helen E6:Elem --> william
Unifier #12
E1:Elem --> helen E2:Elem --> nzwawe E3:Elem --> umugu E4:Elem --> amoc E5:Elem --> taylor E6:Elem --> william
Unifier #13
E1:Elem --> helen E2:Elem --> amoc E3:Elem --> nzwawe E4:Elem --> umugu E5:Elem --> william E6:Elem --> taylor
Unifier #14
E1:Elem --> william E2:Elem --> amoc E3:Elem --> nzwawe E4:Elem --> umugu E5:Elem --> helen E6:Elem --> taylor
Unifier #15
```

```
E1:Elem --> helen E2:Elem --> umugu E3:Elem --> nzwawe E4:Elem --> amoc E5:Elem --> william E6:Elem --> taylor
Unifier #16
E1:Elem --> william E2:Elem --> umugu E3:Elem --> nzwawe E4:Elem --> amoc E5:Elem --> helen E6:Elem --> taylor
Unifier #17
E1:Elem --> taylor E2:Elem --> amoc E3:Elem --> nzwawe E4:Elem --> umugu E5:Elem --> william E6:Elem --> helen
Unifier #18
E1:Elem --> william E2:Elem --> amoc E3:Elem --> nzwawe E4:Elem --> umugu E5:Elem --> taylor E6:Elem --> helen
Unifier #19
E1:Elem --> taylor E2:Elem --> umugu E3:Elem --> nzwawe E4:Elem --> amoc E5:Elem --> william E6:Elem --> helen
Unifier #20
E1:Elem --> william E2:Elem --> umugu E3:Elem --> nzwawe E4:Elem --> amoc E5:Elem --> taylor E6:Elem --> helen
Unifier #21
E1:Elem --> taylor E2:Elem --> amoc E3:Elem --> nzwawe E4:Elem --> umugu E5:Elem --> helen E6:Elem --> william
Unifier #22
E1:Elem --> helen E2:Elem --> amoc E3:Elem --> nzwawe E4:Elem --> umugu E5:Elem --> taylor E6:Elem --> william
Unifier #23
E1:Elem --> taylor E2:Elem --> umugu E3:Elem --> nzwawe E4:Elem --> amoc E5:Elem --> helen E6:Elem --> william
Unifier #24
E1:Elem --> helen E2:Elem --> umugu E3:Elem --> nzwawe E4:Elem --> amoc E5:Elem --> taylor E6:Elem --> william
Unifier #25
E1:Elem --> helen E2:Elem --> nzwawe E3:Elem --> amoc E4:Elem --> umugu E5:Elem --> william E6:Elem --> taylor
Unifier #26
E1:Elem --> william E2:Elem --> nzwawe E3:Elem --> amoc E4:Elem --> umugu E5:Elem --> helen E6:Elem --> taylor
Unifier #27
E1:Elem --> helen E2:Elem --> umugu E3:Elem --> amoc E4:Elem --> nzwawe E5:Elem --> william E6:Elem --> taylor
Unifier #28
E1:Elem --> william E2:Elem --> umugu E3:Elem --> amoc E4:Elem --> nzwawe E5:Elem --> helen E6:Elem --> taylor
Unifier #29
E1:Elem --> taylor E2:Elem --> nzwawe E3:Elem --> amoc E4:Elem --> umugu E5:Elem --> william E6:Elem --> helen
Unifier #30
E1:Elem --> william E2:Elem --> nzwawe E3:Elem --> amoc E4:Elem --> umugu E5:Elem --> taylor E6:Elem --> helen
Unifier #31
E1:Elem --> taylor E2:Elem --> umugu E3:Elem --> amoc E4:Elem --> nzwawe E5:Elem --> william E6:Elem --> helen
Unifier #32
E1:Elem --> william E2:Elem --> umugu E3:Elem --> amoc E4:Elem --> nzwawe E5:Elem --> taylor E6:Elem --> helen
Unifier #33
E1:Elem --> taylor E2:Elem --> nzwawe E3:Elem --> amoc E4:Elem --> umugu E5:Elem --> helen E6:Elem --> william
Unifier #34
E1:Elem --> helen E2:Elem --> nzwawe E3:Elem --> amoc E4:Elem --> umugu E5:Elem --> taylor E6:Elem --> william
Unifier #35
E1:Elem --> taylor E2:Elem --> umugu E3:Elem --> amoc E4:Elem --> nzwawe E5:Elem --> helen E6:Elem --> william
Unifier #36
E1:Elem --> helen E2:Elem --> umugu E3:Elem --> amoc E4:Elem --> nzwawe E5:Elem --> taylor E6:Elem --> william
```

Note that this result is quite impressive, since it cannot be obtained using modern multi-paradigm languages such as Curry [25] and could not be obtained using the `search` command of Full Maude 2.6 with the encoding of [16]. This example requires many relevant features: (i) constraint solving, (ii) logical variables, (iii) order-sorted types, (iv) conditional equations, (v) semantics of values, (vi) strict equality, (vii) associativity for lists, and (viii) associativity, commutativity, and identity for multisets.

7 Conclusions

We have tried to illustrate how the new implementation of variant-based unification available in the most recent Maude 2.7.1 version is paving the way for multi-paradigm programming in Maude. In [16], we considered the *Missionaries and Cannibals* equational logic program of [21] as an example that could not be specified in modern multi-paradigm languages such as Curry [25] but no relevant execution results were included due to the preliminary implementation of narrowing in Maude 2.6. In this paper we have updated the specification and provided more relevant execution results by changing the focus from narrowing with rules (i.e., symbolic reachability) into narrowing with variant equations (i.e., equational unification).

References

1. Alpuente, M., Ballis, D., Falaschi, M.: Transformation and debugging of functional logic programs. In: Dovier, A., Pontelli, E. (eds.) A 25-Year Perspective on Logic Programming. LNCS, vol. 6125, pp. 271–299. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14309-0_13
2. Antoy, S.: Evaluation strategies for functional logic programming. J. Symb. Comput. **40**, 875–903 (2005)
3. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. J. ACM **47**(4), 776–822 (2000)
4. Antoy, S., Hanus, M.: Functional logic programming. Commun. ACM **53**(4), 74–85 (2010)
5. Arts, T., Zantema, H.: Termination of logic programs using semantic unification. In: Proietti, M. (ed.) LOPSTR 1995. LNCS, vol. 1048, pp. 219–233. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-60939-3_17
6. Bae, K., Escobar, S., Meseguer, J.: Abstract logical model checking of infinite-state systems using narrowing. In: van Raamsdonk, F. (ed.) 24th International Conference on Rewriting Techniques and Applications, RTA 2013. LIPIcs, Eindhoven, The Netherlands, 24–26 June 2013, vol. 21, pp. 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
7. Clavel, M., et al.: Maude Manual (Version 2.7.1) (2016). http://maude.cs.illinois.edu
8. Clavel, M., et al.: Unification and narrowing in Maude 2.4. In: Treinen [44], pp. 380–390
9. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
10. Comon-Lundh, H., Delaune, S.: The finite variant property: how to get rid of some algebraic properties. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 294–307. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_22
11. Dershowitz, N.: Goal solving as operational semantics. In: International Logic Programming Symposium, Portland, OR, pp. 3–17. MIT Press, December 1995
12. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Talcott, C.: Associative unification and symbolic reasoning modulo associativity in Maude. In: Rusu, V. (ed.) WRLA 2018. LNCS, vol. 11152, pp. 98–114. Springer, Heidelberg (2018)
13. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Talcott, C.: Built-in variant generation and unification, and their applications in Maude 2.7. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 183–192. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_13
14. Durán, F., Eker, S., Escobar, S., Meseguer, J., Talcott, C.L.: Variants, unification, narrowing, and symbolic reachability in Maude 2.6. In: Schmidt-Schauß [41], pp. 31–40
15. Durán, F., Meseguer, J.: On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. J. Log. Algebr. Program. **81**(7–8), 816–850 (2012)
16. Escobar, S.: Functional logic programming in Maude. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 315–336. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54624-2_16
17. Escobar, S., Meseguer, J.: Symbolic model checking of infinite-state systems using narrowing. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 153–168. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73449-9_13

18. Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. J. Log. Algebr. Program. **81**(7–8), 898–928 (2012)
19. Fay, M.: First-order unification in an equational theory. In: Joyner, W.H. (ed.) Proceedings of the 4th Workshop on Automated Deduction, Austin, Texas, USA, pp. 161–167. Academic Press (1979)
20. Goguen, J., Meseguer, J.: Eqlog: equality, types and generic modules for logic programming. In: DeGroot, D., Lindstrom, G. (eds.) Logic Programming, Functions, Relations and Equations, pp. 295–363. Prentice-Hall, Englewood Cliffs (1986)
21. Goguen, J.A., Meseguer, J.: Equality, types, modules, and (why not?) generics for logic programming. J. Log. Program. **1**(2), 179–210 (1984)
22. Goguen, J.A., Meseguer, J.: Models and equality for logical programming. In: Ehrig, H., Kowalski, R., Levi, G., Montanari, U. (eds.) TAPSOFT 1987. LNCS, vol. 250, pp. 1–22. Springer, Heidelberg (1987). https://doi.org/10.1007/BFb0014969
23. Hanus, M.: The integration of functions into logic programming: from theory to practice. J. Log. Program. **19&20**, 583–628 (1994)
24. Hanus, M.: Multi-paradigm declarative languages. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 45–75. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74610-2_5
25. Hanus, M.: Functional logic programming: from theory to Curry. In: Voronkov, A., Weidenbach, C. (eds.) Programming Logics. LNCS, vol. 7797, pp. 123–168. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37651-1_6
26. Hanus, M.: Multi-paradigm languages. In: Gonzalez, T.F., Diaz-Herrera, J., Tucker, A. (eds.) Computing Handbook, Third Edition: Computer Science and Software Engineering, pp. 66:1–66:17. CRC Press (2014)
27. Hölldobler, S. (ed.): Foundations of Equational Logic Programming. LNCS, vol. 353. Springer, Heidelberg (1989). https://doi.org/10.1007/BFb0015791
28. Hullot, J.-M.: Canonical forms and unification. In: Bibel, W., Kowalski, R. (eds.) CADE 1980. LNCS, vol. 87, pp. 318–334. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10009-1_25
29. Jouannaud, J.-P., Kirchner, H.: Completion of a set of rules modulo a set of equations. SIAM J. Comput. **15**(4), 1155–1194 (1986)
30. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Trans. Program. Lang. Syst. **4**(2), 258–282 (1982)
31. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992)
32. Meseguer, J.: Multiparadigm logic programming. In: Kirchner, H., Levi, G. (eds.) ALP 1992. LNCS, vol. 632, pp. 158–200. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0013826
33. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Presicce, F.P. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-64299-4_26
34. Meseguer, J.: From OBJ to Maude and beyond. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Algebra, Meaning, and Computation. LNCS, vol. 4060, pp. 252–280. Springer, Heidelberg (2006). https://doi.org/10.1007/11780274_14
35. Meseguer, J.: Strict coherence of conditional rewriting modulo axioms. Theor. Comput. Sci. **672**, 1–35 (2017)
36. Meseguer, J.: Symbolic reasoning methods in rewriting logic and Maude. In: Moss, L.S., de Queiroz, R., Martinez, M. (eds.) WoLLIC 2018. LNCS, vol. 10944, pp. 25–60. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-57669-4_2
37. Meseguer, J.: Variant-based satisfiability in initial algebras. Sci. Comput. Program. **154**, 3–41 (2018)

38. Meseguer, J., Thati, P.: Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. High.-Order Symb. Comput. **20**(1–2), 123–160 (2007)
39. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 371–443. Elsevier and MIT Press (2001)
40. Reddy, U.S.: Narrowing as the operational semantics of functional languages. In: Proceedings of the 1985 Second Symposium on Logic Programming, Boston, Massachusetts, 15–18 July 1985, pp. 138–151. IEEE Computer Society Press (1985)
41. Schmidt-Schauß, M. (ed.): Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011. LIPIcs, Novi Sad, Serbia, 30 May–1 June 2011, vol. 10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
42. Slagle, J.R.: Automated theorem-proving for theories with simplifiers commutativity, and associativity. J. ACM **21**(4), 622–642 (1974)
43. TeReSe (ed.): Term Rewriting Systems. Cambridge University Press, Cambridge (2003)
44. Treinen, R. (ed.): RTA 2009. LNCS, vol. 5595. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02348-4
45. Tushkanova, E., Giorgetti, A., Ringeissen, C., Kouchnarenko, O.: A rule-based system for automatic decidability and combinability. Sci. Comput. Program. **99**, 3–23 (2015)
46. Vigneron, L.: Automated deduction techniques for studying rough algebras. Fundamenta Informaticae **33**(1), 85–103 (1998)

MUnit: A Unit Framework for Maude

Adrián Riesco(✉)

Facultad de Informática, Universidad Complutense de Madrid,
Madrid, Spain
ariesco@fdi.ucm.es

Abstract. Unit testing is a widely-used methodology for checking whether the units of a given program work as expected. Maude is a high performance rewriting engine based on rewriting logic. Although Maude has been used to implement complex specifications and tools, it lacks the testing tools usually supported by other languages. In this work we present MUnit, a unit testing framework for Maude that takes into account its main features to define meaningful unit tests. MUnit extends Full Maude and supports functional and system modules, as well as specifications using the Loop Mode and, in particular, Full Maude.

Keywords: Unit testing · Maude · Loop Mode · Full Maude

1 Introduction

Debugging and testing conform one of the most important stages of the software developing cycle, requiring up to the 50% of the time [5]. Among the best known testing methodologies we find property-based testing [8], conformance testing [23], and unit testing [22]. In particular, unit testing is a well-known testing method for checking whether a *unit* behaves as expected. In general, unit tests consist of the unit applied to ground arguments and an expected value; the test passes if both values are equal, although equality can be substituted by a more general notion in particular cases. In imperative languages units usually refer to methods/calls, although other units can be considered depending of the target language. Finally, note also that in unit testing fulfilling the target coverage [3] is in charge of the user.

Maude [10] is a logical framework and high-performance rewriting engine. Maude modules correspond to specifications in *rewriting logic* [15], a logic that allows specifiers to represent many models of concurrent and distributed systems. This logic is an extension of *membership equational logic* [4], an equational logic that, in addition to equations, allows the statement of *membership axioms* characterizing the elements of a sort.

Research partially supported by MINECO Spanish project *TRACES* (TIN2015-67522-C3-3-R) and Comunidad de Madrid project N-Greens Software-CM (S2013/ICE-2731).

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 45–58, 2018.
https://doi.org/10.1007/978-3-319-99840-4_3

Maude modules are executable rewriting logic specifications. Maude functional modules [10, Chap. 4] are executable membership equational specifications that allow the definition of sorts, subsort relations between sorts, operators 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 asserting that terms are equal. Both memberships and equations can be conditional. Maude system modules [10, Chap. 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. Finally, Full Maude [10, Part II] is an extension of Maude written in Maude itself. Full Maude is built on top of the `LOOP-MODE` module [10, 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.

Maude has been used to implement tools such as termination and confluence checkers, theorem provers, real-time extensions, etc. and to specify a wide range of systems, including bio informatics, network protocols, and mobile languages, among many others.[1] However, Maude only provides a limited property-based testing tool [19], which supports functional and system modules. This tool is implemented using narrowing [9], which does not support some theories, in particular those using conditional equations/rules. The transformation to overcome this problem makes the process slower, so the tool lacks the efficiency required to work with large specifications. Since the implementation of MUnit directly uses Maude (meta)commands, the time required to test any Maude specification will be similar to the time required to execute it. More generally, a unit framework is also useful (i) to test functions that do not have associated properties, (ii) to test particular inputs that the user knows might lead to errors, and (iii) to quickly check whether changes in the implementation are correct with respect to a test suite.

In this paper we present MUnit, a unit test framework for Maude that supports functional and system modules, as well as applications on top of the `LOOP-MODE` module (using both input/output facilities and in particular Full Maude features, such as object-oriented modules). Supporting Full Maude applications is particularly interesting for a number of reasons: (i) they are difficult to test, especially those commands producing "side effects" in the internal state of the loop; (ii) in contrast to other Maude applications that are used to analyze particular systems and produce a results, Full Maude applications are designed for being used by other users so, in addition to be thoroughly tested, it is interesting to add the errors reported by users in an easy way; (iii) most of them are large, complex software systems, like Full Maude itself, Real-Time Maude [17], the Maude Formal Environment [11], and the CafeOBJ environment [20], so

[1] See http://maude.cs.illinois.edu/ for a comprehensive list of Maude projects.

a well established testing methodology would help to maintain them; and (iv) there are many applications of this kind; actually, any application requiring I/O interaction or manipulating the database usually extends Full Maude.

The rest of the paper is organized as follows: Sect. 2 presents the main features of MUnit, while Sect. 3 illustrates these features by means of an example. Section 4 outlines the implementation of the tool. Section 5 discusses the related work. Finally, Sect. 6 concludes and proposes some lines of future work to improve the tool. The source code of MUnit, examples, and more information is available at https://github.com/ariesco/MUnit.

2 MUnit

In this section we present the tests available in MUnit. We distinguish the different tests depending on the module under test in order to describe their particularities.

Given that functional modules are confluent and terminating, we can consider that function calls are units that must be reduced to a particular value, and hence equality is enough for checking correctness. Moreover, functional modules support the definition of membership axioms stating the elements of a sort, so we need units to test whether a term has a given sort:

- The test `assertEqual(f(t1, ..., tn), t)` passes if the function `f`, when applied to the ground terms `t1, ...,tn`, is reduced to the same normal form as `t` (modulo axioms). Similarly, the test `assertDifferent(f(t1, ..., tn), t)` passes if the normal form of `f(t1, ..., tn)` is different from the normal form of `t` (modulo axioms). Note that `t` might not be a normal form; for example, it might be a constant defined to ease the testing process.[2] Note also that these tests are commutative, so users can define them according to their preferences.
- MUnit provides shortcuts for Boolean tests. The test `assertTrue(f(t1, ..., tn))` (respectively, `assertFalse(f(t1, ..., tn))`) passes if the term is reduced to `true` (respectively, `false`).
- We can also test whether membership axioms are properly defined by checking the sort of a given term by using `assertSort(t,s)`, which passes if the normal form of the term `t` has exactly sort `s`. MUnit also provides a test `assertLeqSort(t, s)`, which passes if the sort of the normal form of `t` is less or equal to `s`.

When dealing with system modules we face specifications potentially non-terminating and non-confluent. For this reason the units for these modules assert *reachability* rather than equality. Note that object-oriented modules are considered as standard system modules by MUnit.

[2] Likewise, if `f` is not a function but a constructor and we are interested in testing how the terms `t1, ..., tn` behave the test would compute the normal form and compare it with the normal form of `t`. However, this test would not follow completely the philosophy of Maude unit tests as we have defined it.

- The test `assertReachable(t, t')` passes if the term `t'` is reachable from `t` within an unbounded number of steps. Similarly, `assertReachableBnd(t, t', bnd)` adds information about the bound (`bnd`) in the number of steps.
- Usually we are not interested on reaching a specific term but on a solution matching a pattern or just in the lack of solutions. Taking this consideration into account, the test `hasSolution(t, pat, mode, bound, cond)` passes when there exists at least one reachable term that, starting from `t`, matches the pattern `pat` and fulfills the condition `cond` in at most `bound` (either `unbounded` or a natural number) steps. The `mode` can be either `*`, for 0 or more steps; `+`, for 1 or more steps; and `!`, for final states. Similarly, the test `noSolution(t, pat, mode, bound, cond)` passes when no solution was expected.

Finally, we present how to test applications extending the Loop Mode. The basic idea for testing this kind of applications is essentially the same that we have discussed above: testing those units that will be used during the input/output process. However, the internal state of the loop makes some tests difficult to define. For this reason, MUnit provides instructions to start an inner loop and execute commands on it; the user can then perform tests on the intermediate states with the tests above. The instructions for manipulating the inner loop are:

- `loop(initial-state)`. This instruction starts the MUnit inner loop by rewriting `initial-state`, which must have sort `System`. For example, we would use `loop(init)` to start a Full Maude session.
- `command(comm)`. This instruction introduces the command `comm` into the first element of the loop and rewrites the thus obtained term to evolve the system. For example, once started Full Maude with the command above we would use `command(select NAT .)` to change the default module to `NAT`.

Although it is possible to define tests while executing commands, right now we have no means to access the attributes defined in the state (the second argument of the loop). MUnit makes attributes available by using `@` before the attribute name. For example, given that the Full Maude explicit database is identified by `db` we would use `@db` to access it. We can use it to ensure that the database is well formed after introducing new modules, to analyze the modules it contains, and to test functions that require it. In particular, we would check that it is well formed by using `assertSort(@db, Database)`, that indicates that the database has sort `Database` (and hence it is not defined at the kind level, which indicates an error occurred). We show a detailed example in the following section.

3 Running Example

In this section we present how to use the tool by using a simple inventory specification. Note that the tests shown here are used for illustrating the tool; discussions about coverages are beyond the scope of this section. The complete source code of the example is available at https://github.com/ariesco/MUnit.

We first define binary search trees in module BSTREE to store the information about the stock. We will use the product name (a String) as key and a Pair of the form < QTY, PR > as value, where QTY stands for the quantity of the product and PR for its price. Then, we define the sort BSTree for binary search trees and BST? for trees that do not fulfill the appropriate property. Hence, the empty tree (mt) has sort BSTree, while the constructor for bigger trees builds terms of sort BSTree?:

```
fmod BSTREE is
 pr STRING .

 sort BSTree BSTree? Pair .
 subsort BSTree < BSTree? .

 op <_,_> : Nat Nat -> Pair [ctor] .

 op mt : -> BSTree [ctor] .
 op _[_,_]_ : BSTree? String Pair BSTree? -> BSTree? [ctor] .
```

We define a membership axiom to assign the sort BSTree. Given that the left and the right trees have the appropriate sort (i.e., the variables L and R have sort BSTree), we check with the auxiliary function correctOrder that the key in the root is appropriately sorted with respect to the children:

```
 cmb L [S, P] R : BSTree
  if correctOrder(L, S) /\ correctOrder(S, R) .
```

We also specify functions to check whether an item is in the tree (inStock); to insert a new item (insert); to delete an item from the tree (delete); and to update the tree by subtracting one unit of the given item (oneSold):

```
 op inStock : BSTree String -> Bool .
 op insert  : BSTree String Pair -> BSTree .
 op delete  : BSTree String -> BSTree .
 op oneSold : BSTree String -> BSTree .
```

However, we introduced an error in the definition of insert: we did not define the case when the element being introduced is already in the tree:

```
 eq insert(mt, S, P) = mt [S, P] mt .
 ceq insert(L [S, P] R, S', P') = insert(L, S', P') [S, P] R
  if S' < S .
 ceq insert(L [S, P] R, S', P') = L [S, P] insert(R, S', P')
  if S < S' .
 *** eq insert(L [S, P] R, S, P') = L [S, combine(P, P')] R .
 ...
endfm
```

Finally, we define a module FTEST with constants for testing. Tree treeOK1 is a binary search tree with three elements, a, c, and e; treeOK2 consists of treeOK2 after introducing the item b; treeOK3 is treeOK1 after removing the item in the root, c; treeOK4 has one unit of c less than treeOK1; and treeError is a tree with unordered keys:

```
fmod FTEST is
 pr BSTREE .

 ops treeOK1 treeOK2 treeOK3 treeOK4 treeError : ~> BSTree .
 eq treeOK1 = (mt [ "a", < 1, 3 > ] mt) ["c", < 2, 7 > ]
              (mt [ "e", < 3, 5 > ] mt) .
 eq treeOK2 = (mt [ "a", < 1, 3 > ] (mt ["b", < 1, 1 >] mt))
              ["c", < 2, 7 > ] (mt [ "e", < 3, 5 > ] mt) .
 eq treeOK3 = (mt ["a", < 1, 3 >] mt)["e", < 3, 5 >] mt .
 eq treeOK4 = (mt [ "a", < 1, 3 > ] mt) ["c", < 1, 7 > ]
              (mt [ "e", < 3, 5 > ] mt) .
 eq treeError = (mt [ "a", < 1, 3 > ] mt) ["d", < 1, 1 > ]
                (mt [ "c", < 2, 7 > ] mt) .
endfm
```

We test the specification as follows:

- We first check types by using assertLeqSort and assertSort. We can check both the type of particular terms and the sort of the term obtained after applying a function. In our example we check that treeOK1 has sort BSTree? and least sort BSTree. On the other hand, the least sort of treeError is BSTree?, while inserting in a BSTree should return another BSTree. Note that we test how insert behaves with a tree that does not contain the element being inserted (treeOK1) and a tree that contains it (treeOK2).
- Then, we check the obtained results. We use assertTrue and assertFalse to test the Boolean function inStock, while we use assertEqual to indicate how insert, delete, and oneSold should work when applied to treeOK1.

```
(munit FTEST is
  assertLeqSort(treeOK1, BSTree?)
  assertSort(treeOK1, BSTree)
  assertSort(treeError, BSTree?)
  assertSort(insert(treeOK1, "b", < 1, 1 >), BSTree)
  assertSort(insert(treeOK2, "b", < 1, 1 >), BSTree)

  assertTrue(inStock(treeOK1, "a"))
  assertFalse(inStock(treeOK1, "f"))

  assertEqual(insert(treeOK1, "b", < 1, 1 >), treeOK2)
  assertEqual(delete(treeOK1, "c"), treeOK3)
  assertEqual(oneSold(treeOK1, "c"), treeOK4)
endu)
```

When executed, the tool indicates one of the test cases failed. In this case the insertion is not typed as expected because the `insert` function was not completely reduced, since we did not define one of the equations. The output shows how the reduction stopped when facing the missing case:

```
12 test cases were executed.
1 failures.

assertSort(treeOK1,BSTree) passed.

assertLeqSort(treeOK1,BSTree?) passed.

assertSort(treeError,BSTree?) passed.

assertSort(insert(treeOK1,"b",< 1,1 >),BSTree) passed.

assertSort(insert(treeOK2,"b",< 1,1 >),BSTree) failed.
The normal form is (mt["a",< 1,3 >] insert(mt["b",< 1,1 >]mt,"b",< 1,1 >))
                    ["c",< 2,7 >](mt["e",< 3,5 >]mt)
Its sort is BSTree?
...
```

We assume we fix the trees and continue with the example. We use the module above to simulate a shop where buyers try to purchase items and sellers provide new products to the shop. The module `SHOP` defines the sort `People` as a set of `Person`:

```
mod SHOP is
 pr BSTREE .

 sorts Person People Shop .
 subsort Person < People .

 op nobody : -> People [ctor] .
 op __ : People People -> People [ctor assoc comm id: nobody] .
```

Then, we define sellers and buyers. Buyers only take as argument the identifier of the product they want to buy (we assume they buy one unit), while sellers have the identifier of the item, the number of units, and the price they ask for the product. The shop consists of a set of people, the binary search tree storing the available products, and the current money:

```
 *** Product Quantity Price
 op seller : String Nat Nat -> Person [ctor] .
 op buyer : String -> Person [ctor] .

 op [_|_,_] : People BSTree Nat -> Shop [ctor] .
```

Rule `buyer` removes a buyer from the people once he/she buys the item indicated by its argument, given it is available; the shop uses the function `oneSold`

to decrease the quantity of the product and the money of the shop is updated. In turn, the shop obtains more products from sellers with the rule `seller`. This rule indicates that, if the shop has enough money, it adds the products to the inventory, asking buyers to get it for twice the price it paid to the seller:

```
 crl [buyer] : [buyer(S) P | T, M]
  => [P | oneSold(T, S), M + PR]
  if inStock(T, S) /\
     PR := getPrice(T, S) .

 crl [seller] : [seller(S, Q, PR) P | T, M]
  => [P | insert(T, S, < Q, 2 * PR >), sd(M, PR * Q) ]
  if M >= PR * Q .
endm
```

The module `STEST` defines an initial shop with two sellers, offering items `a` and `b`, a `buyer` who wants an item `a`, and `10` as initial money. Note that two different final states are possible: if the shop takes `a` first it will be able to sell it and then it will have money to obtain `b`, while buying `b` first would prevent further rewrites:

```
mod STEST is
 pr FTEST .
 pr SHOP .

 op init-shop : -> Shop .
 eq init-shop = [seller("a", 1, 5) seller("b", 1, 5) buyer("a") | mt, 10] .
endm
```

The tests for this module check that it is possible to reach a final state where `nobody` remains in the shop and it has `5` as money and `"b"` in stock, but it is not possible to reach a state where the money is `20`:

```
(munit STEST is
  hasSolution(init-shop, ([nobody | T:BSTree, 5]), !, unbounded,
              inStock(T:BSTree, "b"))
  noSolution(init-shop, ([P:People | T:BSTree, 20]), +, unbounded, nil)
endu)
```

Instead of simulating the shop we can implement an input/output application extending the Loop Mode for managing it. We would define two commands for buying and adding products:

```
  op sold_ : @Token@ -> @ShopCommand@ [ctor] .
  op add_,_,_ : @Token@ @Token@ @Token@ -> @ShopCommand@ [ctor] .
```

as well as two attributes for storing the current `tree` and `money`:

```
 op tree :_ : BSTree -> Attribute [ctor] .
 op money :_ : Nat -> Attribute [ctor] .
```

The rules dealing with these commands check whether the conditions hold (e.g. there is enough money or the item required is available, respectively) and update the tree and the money accordingly, showing the corresponding message. We show the rule for `add`, which contains a bug and updates the money erroneously (it is decreased when no units are available and unchanged otherwise); we refer to the webpage above for details about the rest of rules and the correct implementation.

```
crl [add] :
    < O : SDC | input : ('add_`,_`,_['token[T], 'token[T'],
                'token[T'']]),  output : nil, tree : BST, money : M, AtS >
 => < O : SDC | input : nilTermList, output : QIL, tree : BST',
                money : M', AtS >
 if Q := downQid(T) /\
    S := string(Q) /\
    QTY := rat(string(downQid(T')), 10) /\
    PR := rat(string(downQid(T'')), 10) /\
    B := M >= QTY * PR /\
    BST' := if B
            then insert(BST, S, < QTY, 2 * PR >)
            else BST
            fi /\
    M' := if B
          then M               *** Should be sd(M, QTY * PR)
          else sd(M, QTY * PR) *** Should be M
          fi /\
    QIL := if B
           then '\n '\! 'Database 'updated. qid(QTY) 'units 'of qid(S)
                'added. '\n
                'It 'was 'bought 'at qid(PR) 'euros. '\n
                'It 'will 'be 'sold 'at qid(2 * PR) 'euros. '\o '\n
           else '\n '\! '\r 'Error: '\o 'Not 'enough 'money. '\n
           fi .
```

In this case we test the specification by starting the loop and first checking that we do not have a `cake` in the inventory and the value of `money` is `10`. Then, we add one cake and pay `5` for it (it will be sold at `10` later). Once the cake has been introduced, we check that in fact the inventory has a `cake` and only `5` as `money`. Then, we indicate that we have sold the cake, hence checking that it disappears from the inventory and we have a final amount of `15` in `money`:

```
(munit SHOP-INIT is
  loop(init-shop)

  assertFalse(inStock(@tree, "cake"))
  assertEqual(@money, 10)

  command(add cake, 1, 5)

  assertTrue(inStock(@tree, "cake"))
```

```
  assertEqual(@money, 5)

  command(sold cake)

  assertFalse(inStock(@tree, "cake"))
  assertEqual(@money, 15)
endu)
```

When executed, the tool indicates that the tests for `@money` fail. The first one states that, after adding a cake, the money is `10` although the user expects it to be `5`. In the second case the money is `20`, which was unexpected but makes sense if we consider the money after introducing the cake was `10`, so the user would check the rule for `add`:

```
6 test cases were executed.
2 failures.
...
command (add cake,1,5) executed.

assertEqual(@money,5) failed.
First term reduced to 10
Second term reduced to 5
...
command (sold cake) executed.

assertEqual(@money,15) failed.
First term reduced to 20
Second term reduced to 15
...
```

4 Implementation

In this section, we present some details of the implementation of MUnit. The source code of the tool is available at https://github.com/ariesco/MUnit.

MUnit extends Full Maude by defining a new type module and its corresponding commands. Following the ideas in [13], the module `MUNIT-SIGNATURE` defines the syntax of MUnit, which will be used later to parse the commands introduced by the user (via its meta-representation in module `META-MUNIT-SIGN`). The module `MUNIT` defines the rule for parsing the only available input in MUnit, MUnit modules:

```
 crl [munit] :
     < O : MUDC | db : DB, input : ('munit_is_endu[T, T']),
                  output : nil, AtS >
  => < O : MUDC | db : DB, input : nilTermList, output : printUR(UR),
                  AtS >
  if UR := procMUnit(T, T', DB) .
```

This rule uses the auxiliary function procMUnit to parse the tests and returns a UnitResult, which consists of a tuple indicating the number of tests that passed and failed, a QidList with the information for the user, and the inner loop:

```
op [_,_,_,_] : Nat Nat QidList Term -> UnitResult [ctor] .
```

The function procMUnit computes the module from the module expression, initializes the loop, and uses the function procProps to traverse the tests in the module and execute them:

```
op procProps : Module Term OpDeclSet Bool Database Term -> UnitResult .
ceq procProps(M, '__[T, T'], VDS, B, DB, SYS) =
                                   [N + N', N1 + N1', QIL QIL', SYS'']
 if [N, N1, QIL, SYS'] := procProps(M, T, VDS, B, DB, SYS) /\
    [N', N1', QIL', SYS''] := procProps(M, T', VDS, B, DB, SYS') .
```

Among all possible tests, we briefly present below how the inner loop is updated. This inner loop consists of a meta-represented loop that is initialized by the loop instruction and is later manipulated when a command instruction is found. The equation below shows (a simplification of) how procProps evaluates the command instruction given the current module, the instruction, and the current loop. It first transforms, in the first two conditions, the command the user wants to introduce into the loop from a list of quoted identifiers into a term. Then, it uses the auxiliary function sysInput to introduce that term as the first argument of the loop. Then, we use metaRewrite to execute the thus obtained loop and generate the information that will be shown to the user. Finally, the result is a tuple containing the tests that passed and failed (0 in both cases, since this instruction is not a test), the message, and the updated loop. Note that other rules are in charge of handling errors in this process.

```
ceq procProps(M, 'command['bubble[T]], SYS) = [0, 0, QIL', SYS'']
 if QIL := downQidList(T) /\
    T' := upTerm(QIL) /\
    SYS' := sysInput(SYS, T') /\
    SYS'' := getTerm(metaRewrite(M, SYS')) /\
    QIL' := '\! 'command '\b ' '`( QIL '`) ' '\o '\! 'executed. '\o '\n .
```

5 Related Work

The maturity of a programming language is, to an extent, related to its tool support. In this way, mainstream languages (mostly imperative) have integrated development environments where debuggers, test tools, integration tools, etc. are integrated, while other languages have limited support of these tools. In fact, as argued in [25], the application of declarative languages out of the academic world is inhibited by the lack of convenient auxiliary tools. However, during the last decades the distance in this subject between declarative and imperative

languages has been reduced thanks to the implementation, among other tools, of debuggers and testing tools adapted to the particular features of these languages. We focus in this section in property-based testing and unit testing, widely used in many declarative languages.

An important step in the field of testing was the development of QuickCheck [14], a property-based testing tool for Haskell. Property-based testing is a two-step technique: first, the user states some properties that the functions under test must fulfill and where some of the inputs have been replaced by so called *generators* for the corresponding data structures. These generators are used during the second step to build several inputs (usually several hundreds) to check whether the property holds. Generators for un-structured datatypes such as natural numbers usually just require to build random elements in a given range, while generators for structured datatypes such as lists are built on top of the generators for their elements, including options for the size of the data structure and the relation between elements.[3] This simple philosophy allows property-based testing tools to check properties on hundreds of inputs in a short time, and experiments have shown that property-based testing is in general as good as more complex testing techniques, so the QuickCheck approach has been implemented in other declaratives languages such as Erlang (PropEr [18] and QuviQ [2]), Scala (ScalaCheck [16]), Curry (EasyCheck [7]), and Prolog (PrologCheck [1]), among many others. In contrast to these tools, the Maude property-based generator tries to falsify the property using narrowing, which allows the tool to find corner cases that can be missed by the tools above but results in a less efficient implementation for big specifications.

Besides property-based testing, several languages support some kind of unit testing. In particular, languages like Erlang (EUnit [6]), Scala (ScalaTest [24]), Prolog (Plunit [26]), and Curry (CurryTest [12]) support this kind of tests. In general, unit tests are used in these languages to state equality between terms. However, the case of Prolog and Curry is interesting in our case because they support non-determinism, just like Maude system modules. In their case they require the user to indicate the set of possible solutions, so each of the solutions are contained in the set.

It is also important to note that property-based testing and unit tests are complementary, as illustrated by the number of languages supporting both approaches. In general, while property-based testing is used to test the main functions, which have associated properties, unit tests are (ideally) defined for checking all functions, so changes in the implementation can be easily checked when integrated.

6 Conclusions and Ongoing Work

We have presented a unit testing framework for Maude. It supports functional and system modules, as well as extensions of the Loop Mode, in particular Full

[3] Most of the tools also include specific strategies for corner cases, such as empty lists.

Maude object-oriented modules and interactive tools. Since Full Maude extensions are in general difficult to test and maintain, one of the main features of MUnit is the support for this kind of applications.

We are currently working to extend MUnit modules with standard Maude declarations. In this way users will be able to define constants and functions to ease the testing process. It would be also interesting to identify tests and support messages in them, so tests would carry information about its intended coverage. It is also interesting to allow users to name tests, so complex tests can be easily identified.

Finally, we are also interested on integrating unit testing and the Maude declarative debugging [21]. Since declarative debugging asks questions about the computation to the user to find the bug, it would generate unit tests that can be later used. Likewise, some answers would be answered by unit tests, saving time and effort to the user.

References

1. Amaral, C., Florido, M., Santos Costa, V.: PrologCheck – property-based testing in prolog. In: Codish, M., Sumii, E. (eds.) FLOPS 2014. LNCS, vol. 8475, pp. 1–17. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07151-0_1
2. Arts, T., Castro, L.M., Hughes, J.: Testing Erlang data types with QuviQ QuickCheck. In: Teoh, S.T., Horváth, Z. (eds.) Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG, pp. 1–8. ACM (2008)
3. Beizer, B.: Software Testing Techniques. Dreamtech, India (2002)
4. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theor. Comput. Sci. **236**, 35–132 (2000)
5. Britton, T., Jeng, L., Carver, G., Cheak, P., Katzenellenbogen, T.: Reversible debugging software. University of Cambridge-Judge Business School, Technical report (2013)
6. Carlsson, R., Rémond, M.: EUnit: a lightweight unit testing framework for Erlang. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, p. 1. ACM (2006)
7. Christiansen, J., Fischer, S.: EasyCheck — test data for free. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 322–336. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78969-7_23
8. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of Haskell programs. In: ACM SIGPLAN Notices, pp. 268–279. ACM Press (2000)
9. Clavel, M., et al.: Maude Manual (Version 2.7), March 2015. http://maude.cs.illinois.edu/w/images/1/1a/Maude-manual.pdf
10. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
11. Clavel, M., Durán, F., Hendrix, J., Lucas, S., Meseguer, J., Ölveczky, P.: The maude formal tool environment. In: Mossakowski, T., Montanari, U., Haveraaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 173–178. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73859-6_12
12. CurryTest. https://www-ps.informatik.uni-kiel.de/currywiki/tools/currytest

13. Durán, F., Ölveczky, P.C.: A guide to extending Full Maude illustrated with the implementation of Real-Time Maude. In: Roşu, G. (ed.) Proceedings of the 7th International Workshop on Rewriting Logic and its Applications, WRLA 2008, Electronic Notes in Theoretical Computer Science, vol. 238(3), pp. 83–102. Elsevier (2009)
14. Hughes, J.: Software testing with QuickCheck. In: Horváth, Z., Plasmeijer, R., Zsók, V. (eds.) CEFP 2009. LNCS, vol. 6299, pp. 183–223. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17685-2_6
15. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992)
16. Nilsson, R.: Scalacheck: The Definitive Guide. IT Pro, Artima Incorporated (2014)
17. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order Symb. Comput. **20**, 161–196 (2007)
18. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: Proceedings of the 2011 ACM SIGPLAN Erlang Workshop, pp. 39–50. ACM Press (2011)
19. Riesco, A.: Using narrowing to test maude specifications. In: Durán, F. (ed.) WRLA 2012. LNCS, vol. 7571, pp. 201–220. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34005-5_11
20. Riesco, A., Ogata, K., Futatsugi, K.: A Maude environment for CafeOBJ. Formal Aspects Comput. **29**(2), 309–334 (2016)
21. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: Declarative debugging of rewriting logic specifications. J. Logic Algebraic Program. **81**(7–8), 851–897 (2012)
22. Runeson, P.: A survey of unit testing practices. IEEE Softw. **23**(4), 22–29 (2006)
23. Tretmans, J.: Conformance testing with labelled transition systems: implementation relations and test generation. Computer Netw. ISDN Syst. **29**(1), 49–79 (1996)
24. Venners, B.: Scalatest 3.0.5 (2018). http://www.scalatest.org/
25. Wadler, P.: Why no one uses functional languages. SIGPLAN Not. **33**(8), 23–27 (1998)
26. Wielemaker, J.: Prolog unit tests. http://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/plunit.html%27)

Parameterized Programming for Compositional System Specification

Óscar Martín(✉), Alberto Verdejo, and Narciso Martí-Oliet

Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain
{omartins,jalberto,narciso}@ucm.es

Abstract. Our overall goal is compositional specification and verification in rewriting logic. In previous work, we described a way to compose system specifications using the operation we call synchronous composition. In this paper, we propose the use of parameterized programming to encapsulate and handle specifications: theories represent interfaces; modules parameterized by such theories instruct on how to assemble the parameter systems using the synchronous composition operation; the implementation of the whole system is then obtained by instantiating the parameters with implementations for the components. We show, and illustrate with examples, how this setting facilitates compositionality.

1 Introduction

Our goal is to provide means for the compositional specification of systems in rewriting logic [12]. In due time, also compositional verification will be addressed. *Compositional* here is in the sense that, first, the design of a complex system is split into simpler components; then, independent specifications are written for each component; and, finally, they are assembled back to form a specification of the whole complex system. In particular, we use Maude [4], a language and system based on rewriting logic.

In our previous paper [10], we defined and studied an operation of *synchronous composition* for rewriting logic specifications, and showed how it can be used for *assembling* component specifications. In the present paper, we show how *parameterized programming* fits in this setting, and provides convenient tools to encapsulate and handle specifications.

To get a taste of what we have already accomplished in [10] and what is coming ahead, let us begin with a very simple example. We want to model two clocks that emit ticks. In principle, each is modeled as an autonomous process, but we are interested in composing them so that their ticks are made simultaneous. The simplest model of such a clock would contain just one rewrite rule:

```
rl [tick] : o => o .
```

Partially supported by Spanish MINECO project TRACES (TIN2015–67522–C3–3–R), and Comunidad de Madrid program N-GREENS Software (S2013/ICE-2731).

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 59–75, 2018.
https://doi.org/10.1007/978-3-319-99840-4_4

We are using the syntax of Maude. The rule represents one transition, called `tick`, looping on one state, called `o`. Given two clocks modeled in this way, say `Clock1` and `Clock2`, their synchronization can be given by a statement like this:

```
sync Clock1 || Clock2 on tick .
```

However, one of the clocks may be quite different:

```
rl [ticking] : a => b .
rl [doingNothingImportantRightNow] : b => a .
```

The transition of interest has now a different label, `ticking`, so that syncing on `tick` is not valid any more. The `sync on` instruction can be adapted to the new situation, but compositionality demands that a module can be internally modified, or even replaced, with no changes needed in the rest of the system. A way to achieve this is that each clock conforms to an *interface* on which the rest of the system can rely. In our case, the interface needs only include a Boolean *property*:

```
op isTicking : -> Ppty{Bool} .
```

This syntax is explained later. The point is that we require that each implementation of a clock provides a definition for the property `isTicking` with an appropriate value for it at each state and transition. To that aim, the first clock specification above needs to be extended with this:

```
eq isTicking @ tick = true .
eq isTicking @ o = false .
```

That is, the property is true while the transition `tick` is being run, false at the state `o`. For the second clock, the definition is:

```
eq isTicking @ ticking = true .
eq isTicking @ G = false [owise] .
```

The keyword `owise` is short for *otherwise*, with the expected semantics; `G` is here a variable ranging over states and transitions. Thus, the property is false at all states and transitions except while the rule `ticking` is being run.

The remaining task is to specify a recipe, or *blueprint*, that, given two clock implementations that conform to the interface, instructs on how to produce their composition:

```
sync Clock1 || Clock2
  on Clock1$isTicking = Clock2$isTicking .
```

The symbol `$` is syntax to qualify a name with the system it belongs to. The criterion for syncing (the `on` clause) involves properties, but no details from the internals of the implementations. Compositionality is preserved in this way.

Our previous work showed the convenience and feasibility of compositionality in rewriting logic. The present one shows how the constructs of parameterized programming, described below, are well-suited to implement interfaces and blueprints and, thus, to support the compositional specification of systems. We begin by giving the needed background on parameterized programming and on the synchronous composition in Sects. 2 and 3. Then, we describe their use for compositional specification in Sect. 4. An example on the alternating bit protocol follows in Sect. 5. Then, we discuss how modular verification fits in this setting in Sect. 6. We finish with related work in Sect. 7 and conclusions in Sect. 8.

2 A Primer on Parameterized Programming and Maude

Parameterized programming is a way to support modularity in programs and specifications. It was introduced in [2] as a feature in the design of the language CLEAR and, then, in the family of OBJ languages, from which Maude took it. Related concepts exist in other languages, bearing names such as generic programming, parametric polymorphism, or type classes. Further information can be found in [4, Sect. 8.3] and [6]. A quick overview follows, using an example from [4].

(A note of warning is necessary. We have chosen to make intensive use of parameterized programming, specially in the example in Sect. 5. It is theoretically plausible, and sticks to the developments in [6]. But not all of it is available in Maude at present. Particularly, parameterized theories and views are only partially implemented in Full Maude, and not at all in Core Maude. See [4] for a description of the features actually available).

There are four kinds of code components in parameterized programming: theories, parameterized modules, implementation modules, and views. A *theory* contains syntactic and semantic requirements. The following describes a theory of strict total orders, called `STOSET`:

```
th STOSET is
   pr BOOL .
   sort Elt .
   op _<_ : Elt Elt -> Bool .
   vars X Y Z : Elt .
   eq X < X = false [nonexec] .  --- irreflexive
   ceq X < Z = true if X < Y /\ Y < Z [nonexec] .  --- transitive
   ceq X = Y if X < Y = false /\ Y < X = false [nonexec] .  --- total
endth
```

The keywords `th` and `endth` enclose the definition of a theory. The keyword `pr` is short for `protecting`: it is a way of importing a module into another. The instruction `pr BOOL` imports the standard Maude module containing the specification for the Booleans, that includes the sort `Bool` and the constants `true` and `false`, all of which are used in the theory.

The two next lines of code contain the syntactic requirements for any module that wants to be an `STOSET`: a sort `Elt` (for *element*) and an infix operator `<`. A module that implements this theory needs not keep these exact names, `Elt` and `<`, because name maps can be defined in views (see below).

The last part of the theory consists of three equations stating the semantic requirements needed so that a module can be considered a strict total order. (Text following three dashes are comments.) Each equation has, attached to it, the `nonexec` attribute, which needs explanation. In Maude modules, equations are used for functional-style programming. Terms are evaluated or reduced by using equations from left to right. In contrast, equations in a theory are not meant to be executed. They represent requirements and, thus, each entails a proof obligation for the programmer who dares to claim that a certain module implements the theory. The `nonexec` attribute prevents Maude from treating them as usual equations.

Natural numbers, with their usual order, are a strict total order. They are implemented in Maude in the standard module NAT that includes, in particular, the sort Nat and the relation <. A *view* is the means to specify in which way an *implementation module* conforms to a theory:

```
view NatAsStoset from STOSET to NAT is
   sort Elt to Nat .
   op _<_ to _<_ .
endv
```

Views are enclosed between the keywords view and endv. Here, NatAsStoset is the name we have chosen for our view. The two lines in the body of the view are name mappings. The expected sort Elt in STOSET is the one defined with the name Nat in the module NAT. The operator's name, <, is the same in the theory and in the implementation. In cases like this, Maude allows to omit the mapping, but we prefer to make it explicit for clarity. Remember that, after having written the view, the obligation is still on us to show that the three equations in STOSET are satisfied in module NAT. We go on considering it proved.

Complex data types, like search trees or lexicographically ordered pairs, can be defined parametric on the underlying strict total order. For our illustrative purposes, we code a *parameterized module* that includes just a function to return the maximum of two given values:

```
fmod MAX{S :: STOSET} is
   op max : S$Elt S$Elt -> S$Elt .
   vars X Y : S$Elt .
   eq max(X, Y) = if X < Y then Y else X fi .
endfm
```

The module is enclosed between the keywords fmod and endfm. It is called MAX and requires a parameter that conforms to the theory STOSET; this formal parameter is called S. The qualified name S$Elt is the way to refer to the sort Elt from the theory S. This qualification is not needed (nor allowed) for operators (like <), because the sorts of the arguments are enough for Maude to know which operator is meant in each case.

Now we can issue Maude commands such as

```
reduce in MAX{NatAsStoset} : max(1, 2) .
```

and expect it to produce a 2. The argument used to feed MAX is not the name of the module NAT, but the view NatAsStoset, that exhibits in which way NAT is an STOSET.

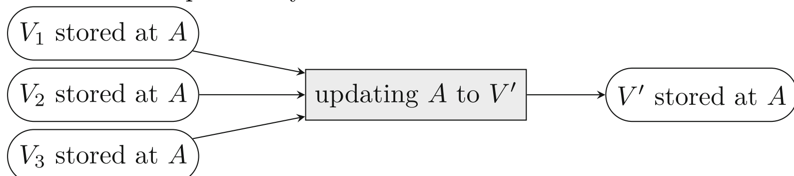
3 A Primer on Synchronous Composition and Egalitarian Maude

In standard rewriting logic there is no way to achieve compositional specification. We mean specifying independent systems and, afterwards, specifying in which way they must be composed and synced. The operation called *synchronous composition* of systems was proposed and formalized in [10] by the same authors of the present paper to mend that weakness. The informal example in the

introduction already showed an instance of this. In this section we explain the main ingredients of our proposal, namely: egalitarianism and properties.

The convenience of putting states and transitions (or actions, or events) at the same level has been claimed many times. See, for instance, the explanations and examples in [8,13]. The specifications of systems and of their temporal properties get clearer or easier when we can refer to both, states and transitions, or to propositions defined on them. We have shown in [10] that the same is true for establishing syncing criteria. However, it is often the case that transitions are treated as second-class citizens. In rewriting logic, in particular, a state is represented by a term of arbitrary complexity, while a rule receives just an atomic label. A state has independent substance, while a transition is given by its origin and destination states (and maybe the label of the rule that produces it).

In [10], we proposed *egalitarian* systems and structures. In them, transitions have meaning by themselves, and the same transition can have several origin and destination states. For example, if modeling a memory storage, the updating of a given memory address A to whatever new value V' does not depend on the value previously stored:



Each run of this system would involve a different origin state, but the transition is the same in all cases.

Being egalitarian is not fundamental to our approach, and our developments make sense in a state-only setting as well. But we find it convenient, and we stick to it in this paper.

Syntactically, egalitarian Maude modules differ from standard ones in that rule labels are given by complex terms, in the same way as states are. The choice of label term and, in particular, the variables in it, have semantic consequences. See [10,11] for more on this.

When the time comes for specifying criteria to sync components, we do not want them to depend on the internal details of the implementation of the component modules. Instead, we require and assume that *properties* are defined as an extension to the component specification. Properties work like functions defined at states and at transitions. Their return data type can be whatever is appropriate in each case. In a way, properties can be seen as a generalization of the propositions used in labeled transition systems, and in Maude for LTL model checking purposes. In the example from the introduction, `isTicking` is a Boolean property, equivalent to a proposition.

In egalitarian Maude, syncing criteria are written as equalities between properties from two systems, like `Clock1$isTicking = Clock2$isTicking` or, in general, `S1$P1 = S2$P2`. We use the `$`-syntax to show which property is meant in each case. More than one such criteria can be required to form a composed system. We refer again the reader to the example in Sect. 5.

Properties get values at states and at transitions, and it can very well happen that a state in a component system needs to be visited at the same time as a transition at the other. In the clocks example in the introduction, the state `o` in the first clock needs to sync with both states in the second clock, `a` and `b`, and to the non-ticking transition between them. Thus, in a composed system we can rarely talk about pure states or pure transitions. We use the name *stage* to refer to any combination of individual states and transitions from different components.

Syncing can be used to ensure simultaneity, as for the clocks in the example in the introduction. But it also allows to emulate the passing of data from one process to another. Consider again a memory that has values stored at addresses. The specification of this memory must include a wild non-determinism from each state, allowing it to update any address to any value. But suppose that there is another component modeling a processor, and that it is in need to store in the memory a particular result of some internal computation. The memory's updating has to be synced to the processor's action, requiring agreement on the updating value and address. This reduces the possible choices of the memory to just one as *commanded* by the processor. The result is equivalent to the processor *passing* address and value to the memory. We make intensive use of this technique in the example in Sect. 5.

Our proposals entail a proper extension of Maude's language, so the execution engine and other tools in the Maude system cannot be used on specifications containing synchronous composition or egalitarian features. In [10] we present an operation that we call *split* that transforms one of our specifications into an *equivalent* one in standard Maude. The precise sense of that equivalence is described in the cited paper. We include next a glimpse of the workings of the split operation.

First, each rewrite rule is split into two unlabeled *half rules*, for example:

```
rl [ticking] : a => b .    --split-->    rl a => ticking .
                                         rl ticking => b .
```

In this way, each transition is transformed into a new state. Then, rules from different modules are composed, with syncing criteria put in place:

```
rl o => tick .
                  ||  -->   crl < o, a > => < tick, ticking >
rl a => ticking .             if Clock1$isTicking @ tick
                               = Clock2$isTicking @ ticking .
```

The condition in the resulting rule is the syncing criterion `Clock1$isTicking = Clock2$isTicking` applied to the destination stages. In some cases, like in the example, the condition is always true, and the rule can be simplified to a non-conditional one.

4 Parameterized Programming for System Specification

We propose here the use of theories (in the parameterized-programming sense) to specify interfaces to which component implementations must conform. Such theories, therefore, contain declarations of properties and their sorts. Concrete implementations must contain, in addition to the internal workings of the system, the actual definitions for such properties and sorts. Then, composed systems are specified in parameterized modules that take the components as parameters and contain a `sync on` instruction. In this section, we fully formalize the clock example from the introduction, and use it to guide the explanations.

To begin with, this is a module containing some declarations useful to code atomic systems:

```
fmod STAGES is
   sorts State Trans Stage .
   subsorts State Trans < Stage .
   op init : -> Stage .
endfm
```

Terms of sorts `State` and `Trans` represent states and transitions. The sort `Stage` includes both of them. We have chosen the name `init` for the initial stage, which can be a state, but also a transition (as if we start analyzing the system when it is already *doing* something).

Whenever we need a property sort, we create it with this parameterized module:

```
fmod PPTY{X :: TRIV} is
   pr MAYBE{X} .
   sort Ppty{X} .
   sort Stage .
   op _@_ : Ppty{X} Stage -> Maybe{X} .
endfm
```

There are several points here that may need explanation. The theory `TRIV` used in the argument is defined in Maude as:

```
th TRIV is
   sort Elt .
endth
```

That is, to conform to `TRIV` a module just needs to provide a sort (to replace `Elt`). It is a way to pass a sort to a module. In this way, we can create properties of any given sort.

The parameterized module `MAYBE` is built-in in Maude, although we use our own slightly different definition:

```
fmod MAYBE{X :: TRIV} is
   sort Maybe{X} .
   subsort X$Elt < Maybe{X} .
   op none : -> Maybe{X} .
endfm
```

Thus, the parameterized sort `Maybe{X}` includes all the values in the parameter sort `X$Elt` plus a new element called `none`. This `none` element is often needed as a dummy or special value for properties, as shown in the example in Sect. 5.

In the body of the module `PPTY`, the parameterized sort `Ppty{X}` is declared as well as the sort `Stage`, and the infix operator `@`, that evaluates a property at a stage and returns either a value in `X$Elt` or `none`. For example, we can instantiate to `PPTY{Bool}`, using the built-in view `Bool` that maps `Elt` to the sort `Bool` of Booleans. This would produce declarations for the sort `Ppty{Bool}` and the appropriately typed operator `@`.

The following is our first example of a theory to be used as system interface, namely, for clocks:

```
th CLOCK-IF is
   pr PPTY{Bool} .
   op isTicking : -> Ppty{Bool} .
endth
```

Thus, `isTicking` is a property that, when evaluated at a stage, produces (maybe) a Boolean. We include `-IF` as a suffix to all names for interfaces. Thus, a clock is anything able to inform us on whether it is ticking or not at each stage.

We can now specify how two clocks conforming to `CLOCK-IF` are composed and synced:

```
emod TWO-SYNCED-CLOCKS-BP{Cl1 :: CLOCK-IF, Cl2 :: CLOCK-IF} is
   sync Cl1 || Cl2
      on Cl1$isTicking = Cl2$isTicking .
endem
```

We call this kind of modules *blueprints*, and give them names ending in `-BP`. The instruction `sync on` is not standard Maude, but part of the *egalitarian Maude* extension we intend to implement soon. Its semantics should be clear by now.

The blueprint module is enclosed between the keywords `emod` and `endem`. The `e` stands for *egalitarian*. Blueprint modules must contain a `sync on` instruction, and are not allowed to contain new rules, nor to tinker with `Stage` terms (like producing new ones or identifying existing ones). Only *atomic* rewrite systems have rules; a composed system evolves according to the rules of its atomic components (or subcomponents) and the syncing criteria provided. The only code allowed in a blueprint module, in addition to the `sync on` instruction, is whatever is needed to declare and define the properties of the composed system, in case it is going to be used as a component in turn. Those properties must be defined in terms of the properties of the components, and not relying on internal implementation details. All these choices are meant to promote modularity in specification and design.

It is time for concrete implementations. This is the simple clock from the introduction, fully specified:

```
aemod CLOCK is
   pr STAGES .
   op o : -> State .
   op tick : -> Trans .
   rl [tick] : o => o .
   eq init = o .
endaem
```

Terms of sort `Trans` are used as rule labels. Ground terms of sort `Trans` represent transitions. Constructors for `Trans` need to be declared, as well as for `State`.

In this simple example rule labels are constants, but in general they can be terms of any complexity, the same as for states.

The clock module is enclosed between the keywords `aemod` and `endaem`. The `a` stands for *atomic*, that is, not composed, and the `e` stands for *egalitarian*. These modules are very similar to standard system modules in Maude, the main difference being, as already noted, that rule labels are terms of sort `Trans`.

That module models the inner workings of the clock, but does not conform to the `CLOCK_IF` interface as yet. We fulfill this need by extending the module to define the property:

```
aemod CLOCK-PPT is
   pr CLOCK .
   pr PPTY{Bool} .
   op isTicking : -> Ppty{Bool} .
   eq isTicking @ tick = true .
   eq isTicking @ o = false .
endaem
```

We give property modules names ending in `-PPT`.

The needed view is now simple:

```
view Clock from CLOCK-IF to CLOCK-PPT is
   op isTicking to isTicking .
endv
```

We have chosen the same name, `isTicking`, for the property defined in the implementation `CLOCK-PPT` and for the one declared in the interface `CLOCK-IF`, but it certainly needs not be so, because name mappings are specified in the view. Indeed, when two systems compose, it is common that the property on which they must sync is seen in a different way from each side, and different names are appropriate, like in, say, `isMessageSent` at one side and `isMessageReceived` at the other.

The final implementation of the complete system is obtained by feeding the blueprint with actual implementations of the components, that is, with the expression `TWO-SYNCED-CLOCKS-BP{Clock, Clock}`.

Composed systems can also be used as implementations, conforming to an interface. For instance, two clocks with simultaneous ticks can be considered and used as a single clock. To that aim, the composed clock needs to inform about its ticks:

```
emod TWO-SYNCED-CLOCKS-BP-PPT{Cl1 :: CLOCK-IF, Cl2 :: CLOCK-IF} is
   pr TWO-SYNCED-CLOCKS-BP{Cl1, Cl2} .
   pr PPTY{Bool} .
   op areBothTicking : -> Ppty{Bool} .
   var G : Stage .
   eq areBothTicking @ G = isTicking @ Cl1(G) .
endem
```

A side effect of our synchronous composition operation is the definition of projection operators: `Cl1(G)` is the part of the stage `G` that belongs to the component `Cl1`. Again, there is more on this in [10,11]. In this particular case, using `Cl2` instead of `Cl1` would do the same.

We claim that this can be used as a single clock:

```
view 2ClocksAsAClock{Cl1 :: CLOCK-IF, Cl2 :: CLOCK-IF}
  from CLOCK-IF to TWO-SYNCED-CLOCKS-BP-PPT{Cl1, Cl2} is
  op isTicking to areBothTicking .
endv
```

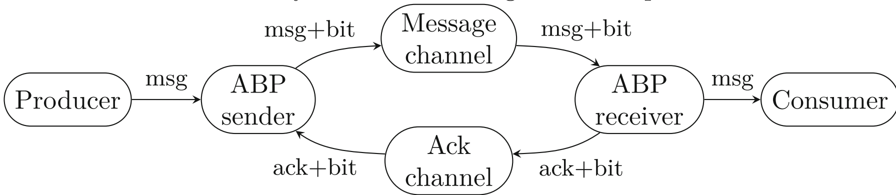
This is a parameterized view, valid for any pair of clock implementations `Cl1` and `Cl2`. The following is a valid module expression for a system of 1+2 clocks:

```
TWO-SYNCED-CLOCKS-BP{Clock, 2ClocksAsAClock{Clock, Clock}}
```

5 Example: Alternating Bit Protocol

The alternating bit protocol, or ABP, is used to transmit messages on a lossy channel, that is, a channel that can lose some of the messages it receives before delivering them at the other side. According to ABP, a bit is attached to each packet of information sent through the channel. The sender must keep on sending the same packet with the same bit until it receives an acknowledgement from the receiver with that same bit. Then, the sender starts sending the next packet with the bit inverted. As also acknowledgements can be lost in the channel, the receiver must keep on acknowledging until it receives a new message, with a different bit, that suffices as proof that its acknowledgement was received and processed by the sender.

We consider an ABP system as consisting of six components:



At the two ends there are a producer and a consumer, that do not care about communication protocols. The sender and the receiver are the components that implement the ABP. There are two channels: one for transmitting messages originating in the producer; the other for transmitting acknowledgements. We call *message* to whatever the producer wants to transmit to the consumer. The pieces of information that are sent through a channel are called *packets*. It is the sender's task to transform each message into one or more packets; the receiver has the inverse task. What precisely is a packet depends on the particular implementation of the protocol. In our implementation of ABP, we make the rather unrealistic assumption that we can build channels capable of transmitting packets of whatever data type. Our implementation of channels is parametric on that data type.

The six modules in our diagram are not all of the same nature. The producer, the consumer, and the two channels can be seen as representing physical entities, while the sender and the receiver may well be pieces of software running on some of them. This difference, however, has no role in our specifications.

In the rest of this section we translate the diagram to interfaces and blueprints, that is, to `th`s and `emod`s. We are not interested here in the actual implementations of the components (the `aemod`s) but we include below the one for the channels as an example. The complete code is described in detail in [11].

As explained in Sect. 3, properties can be used, in particular, to emulate value passing through in and out ports. The direction of the arrows in the diagram shows the use we intend to make of properties. But properties are not directional in essence. Thus, it happens that the producer and the consumer conform to the same interface:

```
th PROCESS-IF{Msg :: TRIV} is
   pr PPTY{Msg} .
   op msgMoving : -> Ppty{Msg} .
endth
```

We use the term *moving* to be agnostic about the direction followed by the message, in or out. The expected value of `msgMoving` is the message being given out by the producer, or taken in by the consumer. At any other moment, or stage, when such movements are not taking place, the value of `msgMoving` has to be `none`. The same behavior is expected from corresponding properties of the sender and the receiver, to be discussed later. This fulfills the need of value passing (message passing, in this case) and also achieves simultaneity of actions, because only stages related to value passing have `msgMoving` different to `none`.

We have made this interface (and others) parametric on the sort of messages (or packets), so that it is valid, no matter what the producer and the consumer need to interchange. Thus, we are using parameterized programming in two ways: the standard use, with theories like `TRIV`, and the interface use, like `PROCESS-IF`.

The two pieces that implement the ABP, sender and receiver, also share interface:

```
th PROTOCOL-IF{Msg :: TRIV, Pck2Chnl :: TRIV, PckFChnl :: TRIV} is
   pr PPTY{Msg} + PPTY{Pck2Chnl} + PPTY{PckFChnl} .
   op procMsgMoving : -> Ppty{Msg} .
   op pckLeaving2Chnl : -> Ppty{Pck2Chnl} .
   op pckComingFChnl : -> Ppty{PckFChnl} .
endth
```

A protocol accepts three arguments: the sort of the messages it receives from the producer or delivers to the consumer, the sort of the packets it sends through the channel, and the sort of the packets it receives from the other channel. Also, there are three properties needed in a protocol, that correspond to the three arrows that arrive to or depart from the sender and the receiver in the diagram. This interface does not presuppose any particular relation between messages and packets, so that it is valid for different protocols. In our implementation of ABP, there are two kinds of packets: message packets, that contain a message and the alternating bit; and acknowledgement packets, whose only relevant information is the alternating bit. Implementations of senders and receivers instantiate each parameter appropriately.

Finally, the interface for the channels:

```
th CHANNEL-IF{Pck :: TRIV} is
   pr PPTY{Pck} .
   op pckComing pckLeaving : -> Ppty{Pck} .
endth
```

It is parametric on the sort of packets. The two properties have the meaning that can be expected from their names. As always, interfaces are independent of internal behavior: an implementation of channels that transmit a packet at a time is as valid as one able to keep a queue of packets to be delivered in turn.

The implementation of a channel is rather straightforward. We include it next, particularly to illustrate the way properties are defined. A channel can just accept a packet, deliver it, or lose it:

```
aemod CHANNEL{Pck :: TRIV} is
   pr STAGES .
   pr MAYBE{Pck} .
   subsort Maybe{Pck} < State .
   ops acceptingPck deliveringPck : Pck$Elt -> Trans .
   op losingPck : -> Trans .
   var P : Pck$Elt .
   crl [acceptingPck(P)] : none => P if "P is a possible packet" .
   rl [deliveringPck(P)] : P => none .
   rl [losingPck] : P => none .
   eq init = none .
endaem
```

The condition for the first rule presents some subtleties that we prefer to avoid here. Based on that implementation, we declare and define properties:

```
aemod CHANNEL-PPT{Pck :: TRIV} is
   pr CHANNEL{Pck} .
   pr STAGES .
   pr PPTY{Pck} .
   op pckComing pckLeaving : -> Ppty{Pck} .
   var P : Pck$Elt .
   var G : Stage .
   eq pckComing @ acceptingPck(P) = P .
   eq pckComing @ G = none .
   eq pckLeaving @ deliveringPck(P) = P .
   eq pckLeaving @ G = none .
endaem
```

We claim that our implementation is indeed a channel, that is, that it conforms to `CHANNEL-IF`:

```
view Channel{Pck :: TRIV}
   from CHANNEL-IF{Pck}
   to CHANNEL-PPT{Pck} is
   op pckComing to pckComing .
   op pckLeaving to pckLeaving .
endv
```

Instead of composing all our six components at once, we prefer to encapsulate the sender, the receiver, and the two channels, to produce a communication

system to which a producer and a consumer can be attached later. So this is the blueprint for a communication system:

```
emod COMM-SYSTEM-BP
   { Sndr :: PROTOCOL-IF{Msg :: TRIV,
                         MsgPacket :: TRIV,
                         AckPacket :: TRIV},
     MsgChnl :: CHANNEL-IF{MsgPacket :: TRIV},
     AckChnl :: CHANNEL-IF{AckPacket :: TRIV},
     Rcvr :: PROTOCOL-IF{Msg :: TRIV,
                         AckPacket :: TRIV,
                         MsgPacket :: TRIV}
   } is
   sync Sndr || MsgChnl || AckChnl || Rcvr
      on Sndr$pckLeaving2Chnl = MsgChnl$pckComing
      /\ MsgChnl$pckLeaving = Rcvr$pckComingFChnl
      /\ Rcvr$pckLeaving2Chnl = AckChnl$pckComing
      /\ AckChnl$pckLeaving = Sndr$pckComingFChnl .
endem
```

The list of arguments is long, but not difficult to grasp. There are four arguments, one for each component system. Sender and receiver conform to the `PROTOCOL-IF` interface; the two channels to `CHANNEL-IF`. Each of these interfaces is given by a parameterized theory, and we need to give formal names to their parameters. Coincidence of names is used to make explicit the sorts that are shared between different components. For instance, the same `Msg` is a parameter of both the sender `Sndr` and the receiver `Rcvr`. The four syncing criteria correspond closely to the four arrows joining these components in the diagram at the beginning of this section. Each criterion states that values that leave a component come to another. As explained before, qualification with the `$`-syntax allows disambiguation for property names.

This `COMM-SYSTEM-BP` is still a quite general recipe, probably useful to implement many acknowledgement-based communication protocols. In our final implementation of ABP, the only free choice left is the message sort given by the parameter `Msg`; packets are built from it by adding the alternating bit. Thus, an ABP communication system is obtained like this:

```
emod ABP-SYSTEM{Msg :: TRIV} is
   pr COMM-SYSTEM-BP { AbpSender{Msg,
                                 Packet{Msg, Bool},
                                 Packet{Ack, Bool}},
                       Channel{Packet{Msg, Bool}},
                       Channel{Packet{Ack, Bool}},
                       AbpReceiver{Msg,
                                   Packet{Ack, Bool},
                                   Packet{Msg, Bool}} } .
endem
```

A few remarks are needed on the views used and the modules they refer to. The sender and the receiver work differently and need independent implementations, with views `AbpSender` and `AbpReceiver`. But the two channels work the same, so that they can share a single implementation and view, with their parameter instantiated to transmit either message packets or acknowledgement ones.

The view `Packet` refers to a module used to build a packet from two elements, one from each of its argument sorts.

Interestingly, any such composed system can be viewed as a channel in itself—a kind of trustworthy channel. Hiding some boring parts:

```
view CommSystemAsChannel{Msg :: TRIV} is
   from CHANNEL-IF{Msg}
   to COMM-SYSTEM-BP-PPT{Sndr..., MsgChnl..., AckChnl..., Rcvr...} is
   op pckComing to msgMovingSndr .
   op pckLeaving to msgMovingRcvr .
endv
```

This is assuming that two properties have being declared and defined in an omitted module `COMM-SYSTEM-BP-PPT` extending `COMM-SYSTEM-BP`.

6 Guarantees, Assumptions, and Compositional Verification

A lossy channel and a trustworthy channel are not the same thing. Syntactically, they conform to the same interface, but their behaviors are different. In the same way that we need to include `nonexec` equations in the theory `STOSET` to make it actually represent strict total orders, we need a way to specify the expected behavior of systems. And, while equations are appropriate for algebraic requirements and static data structures, a good choice for the behavior of reactive systems are formulas in some temporal logic. In this paper, we use LTL, for which Maude has got a model checker.

For example, for a lossy channel, we need to require that at least some packets keep getting through it:

```
th LOSSY-CHANNEL-IF{Pck :: TRIV} is
   inc CHANNEL-IF{Pck} .
   grt [] <> pckComing =/= none -> [] <> pckLeaving =/= none .
endth
```

The new keyword `grt`, short for *guarantees*, introduces an LTL formula whose text can use properties and values already declared in the theory. The bare name of a property is used to represent the different values this property takes in the stages the system visits during its execution.

Each purported implementation of this theory gets thus a proof obligation. In some cases, the proof can be obtained by using Maude's model checker.

In addition to guarantees, we may need to specify *assumptions*, that is, behaviors from the environment that any implementation of the interface can take for granted. For instance, if a non-ending provision of coming packets is appropriate, this line can be added to the theory:

```
   ass [] <> pckComing =/= none .
```

Those temporal formulas enrich the interfaces with semantic requirements. But *assume-guarantee* is also a well-known paradigm for the compositional verification of reactive systems [9]. The idea is that we can draw conclusions on the behavior of the whole system from the assumptions and guarantees of each

component, even if no implementation is available yet. We plan to go deeper into this issue in a future paper.

7 Related Work

The idea of interfaces for encapsulation is well known in computer science, and is present both in mainstream languages and in formal ones, in one flavor or another. Compositionality is also a widespread concept, be it called synchronous product or method invocation. In this sense, our proposals are related to theoretical developments such as Span(Graph), interface theories, and coordination, among others (they are described, respectively, in [1,7,15]). These works put an emphasis on separation of concerns between implementation and composition, made possible by some kind of interfaces, often using algebraic language.

A discussion of these works belongs, however, elsewhere; namely, to the paper [10] (and its eventual journal realization), where we describe in full detail our concepts of composition and interface. The present paper is instead focused on the use of parameterized programming and the possibility of compositional verification. No system or theoretical development we know of includes all such features. We quickly review next three that come close: nuXmv, PVS, and CafeOBJ.

nuXmv is described in [3]. It is the latest version of the model checker that started as SMV. It includes constructs for data interchange and syncing between processes, but no concept of abstract interface. Temporal formulas representing guarantees, either in LTL or CTL, are embedded as part of the specification of a system, and the tool model checks the formulas against the implementation. No place for assumptions, although this latest version seems to include commands for modular verification.

PVS is mainly a theorem prover, but it includes facilities for reactive system specification and model checking. Theories can be used as interfaces, the same as in Maude, as shown in [14], but it is not possible to use them for system composition, and compositional verification is not supported. All the information is on the PVS website at http://pvs.csl.sri.com.

CafeOBJ is the tool that comes closer to our wishes. Information is available on its website at https://cafeobj.org. It allows specifications based on *hidden algebra*, that is, that some sorts and operations are hidden, representing the internal state of the system, while others can be observed. These *observables* correspond to our properties. Some kind of compositionality is possible based on hidden algebra, as described, for example, in [5], although the components are not really independent entities, but are specified all together, with no synchronous composition operation. In our approach, we do not insist, as hidden algebra does, that a state (or stage) be determined by its properties: we just define the properties that are needed for syncing, or for writing LTL formulas. CafeOBJ includes facilities for searching and for theorem proving.

8 Concluding Remarks

Parameterized programming is a well-known paradigm. The synchronous composition operation of rewrite systems was discussed in [10]. In the present paper, we show how it all fits together to make compositional specification and verification possible within rewriting logic. This includes the use of temporal-logic theories as interfaces, and modules parameterized by such theories as blueprints for specifying composition. As future work, our to-do list includes the implementation on Maude of the synchronous composition operation and the syntactic extensions needed to allow for LTL formulas in theories. The missing piece that would complete the jigsaw is a tool for deducing temporal properties for the composed system based on the temporal properties of the components.

While the standard use of parameterized programming, described in [4] and in Sect. 2 of this paper, is appropriate for functional modules and for the specification of static data structures, we dare to assert that our use of it for parameterized specification is the appropriate one for reactive systems.

Acknowledgments. We thank the anonymous referees for their careful and clever remarks that helped us to improve this paper.

References

1. de Alfaro, L., Henzinger, T.A.: Interface-based design. In: Broy, M., Grünbauer, J., Harel, D., Hoare, T. (eds.) Engineering Theories of Software Intensive Systems, pp. 83–104. Springer, Dordrecht (2005). https://doi.org/10.1007/1-4020-3532-2_3
2. Burstall, R.M., Goguen, J.A.: The semantics of CLEAR, a specification language. In: Bjøorner, D. (ed.) Abstract Software Specifications. LNCS, vol. 86, pp. 292–332. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10007-5_41
3. Cavada, R.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
4. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
5. Diaconescu, R., Futatsugi, K., Iida, S.: Component-based algebraic specification and verification in CafeOBJ. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1644–1663. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48118-4_37
6. Durán, F., Meseguer, J.: Parameterized theories and views in Full Maude 2.0. In: Futatsugi, K. (ed.) WRLA 2000. Electronic Notes in Theoretical Computer Science, pp. 316–338. Elsevier, Amsterdam (2000). https://doi.org/10.1016/S1571-0661(05)80136-7
7. Gianola, A., Kasangian, S., Sabadini, N.: Cospan/Span(Graph): an algebra for open, reconfigurable automata networks. In: CALCO. LIPIcs, vol. 72, pp. 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/LIPIcs.CALCO.2017.2

8. Kindler, E., Vesper, T.: ESTL: a temporal logic for events and states. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 365–384. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-69108-1_20
9. Kupferman, O., Vardi, M.Y.: An automata-theoretic approach to modular model checking. ACM Trans. Program. Lang. Syst. **22**(1), 87–128 (2000). https://doi.org/10.1145/345099.345104
10. Martín, Ó., Verdejo, A., Martí-Oliet, N.: Modular specification in rewriting logic (extended version). Technical report 04/17, Departamento de Sistemas Informáticos y Computación, Facultad de Informática, Universidad Complutense de Madrid (2017). http://eprints.ucm.es/45264/
11. Martín, Ó., Verdejo, A., Martí-Oliet, N.: Alternating bit protocol as an example of compositional system specification. Technical report 01/18, Departamento de Sistemas Informáticos y Computación, Facultad de Informática, Universidad Complutense de Madrid (2018). http://eprints.ucm.es/46243/
12. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992). https://doi.org/10.1016/0304-3975(92)90182-F
13. Meseguer, J.: The temporal logic of rewriting: a gentle introduction. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 354–382. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68679-8_22
14. Owre, S., Shankar, N.: Theory Interpretations in PVS. Techical report, SRI International, April 2001 (2001). http://pvs.csl.sri.com/doc/interpretations.pdf
15. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. Adv. Comput. **46**, 329–400 (1998). https://doi.org/10.1016/S0065-2458(08)60208-9

Symbolic Specification and Verification of Data-Aware BPMN Processes Using Rewriting Modulo SMT

Francisco Durán[1(✉)], Camilo Rocha[2], and Gwen Salaün[3]

[1] Universidad de Málaga, Málaga, Spain
duran@lcc.uma.es
[2] Pontificia Universidad Javeriana, Cali, Colombia
camilo.rocha@javerianacali.edu.co
[3] Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, 38000 Grenoble, France
gwen.salaun@inria.fr

Abstract. The *Business Process Model and Notation* (BPMN) is the standard notation for modeling business processes. It relies on a workflow-based language that allows for the modeling of the control-flow graph of an entire process. In this paper, the main focus is on an extension of BPMN with data, which is convenient for describing real-world processes involving complex behavior and data descriptions. By considering this level of expressiveness due to the new features, challenging questions arise regarding the choice of the semantic framework for specifying such an extension of BPMN, as well as how to carry out the symbolic simulation, validation, and correctness of the process models. These issues are addressed first by providing a symbolic executable rewriting logic semantics of BPMN using the rewriting modulo SMT framework, where the execution is driven by rewriting modulo axioms and by querying SMT decision procedures for data conditions. Second, reachability properties, such as deadlock freedom and detection of unreachable states with data exhibiting certain values, can be specified and automatically checked with the help of Maude, thanks to its support for rewriting modulo SMT. The approach presented in this paper has been validated on realistic processes and it is illustrated with a running example.

1 Introduction

Business processes are omnipresent in companies all around the world. A business process is a collection of structured activities or tasks that produce a specific product and fulfil a specific organizational goal for a customer or market. A process aims at modeling activities, their causal and temporal relationships, and specific business rules that process executions have to comply with. In this context, business process modeling is of prime importance to analyze and control business processes. With appropriate techniques, these models may help to the improvement of such processes.

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 76–97, 2018.
https://doi.org/10.1007/978-3-319-99840-4_5

Business processes are usually described using workflow-based notations. BPMN is one of these notations. It was normalized by ISO/IEC in 2013 [15] and has become the *de facto* standard for modeling business processes. BPMN is a quite expressive notation that describes the order in which a set of activities is executed. Beyond basic operators (e.g., beginning, end, sequence), the notion of gateways allows designers to specify different evolutions of the process control flow (e.g., choice, parallel, message-based). In this work, there is the particular interest on data descriptions, which take two different forms in BPMN processes, namely: (i) as variables that are initialized and modified during the process execution, and (ii) as conditions that may be used in gateways to decide the branches to be triggered at runtime. Representing data and conditions in BPMN is crucial for enabling the modeling of real-world processes, where data is pervasive and comes from different sources. BPMN tools such as the Activiti or Bonita platforms provide support for the definition of data-aware workflows. However, as it can be seen in the related work section of this paper, just a few formal specification and verification approaches consider this level of expressiveness.

The research presented in this paper takes a step farther by supporting the symbolic specification, execution, and analysis of data-aware processes with an external and nondeterministic environment (e.g., interaction with user input or sensor probing). This basically means that process variables can be initialized using assignments or by means of lookup operations, so that their values are nondeterministically provided by the environment. The consideration of such open systems, however, poses new challenges that include the potential infinitely-branching nondeterminism due to the environment. Queries over such systems are, in general, beyond the reach of ground rewriting and would require, for instance, inductive techniques over the rewrite relation.

Given such a general and symbolic modeling language for BPMN processes, there are several questions that arise from a correctness point of view. For example, is it possible to verify that a given process is deadlock free for any possible input and interaction with the environment? Are there parts of the workflow that are never reached or cannot be reached during execution for certain initial conditions? Are there possible executions of the process leading to a state where the variables have specific values? It is not easy to answer these questions when considering data ranging over possibly infinite domains. Indeed, most approaches on the analysis and verification of workflows are based on process over-approximation in which data is abstracted, either just by removing it or by replacing it with stochastic information. Although these approaches are useful in the formal analysis of workflows, they can miss important information hidden in the data. E.g., they can miss deadlocks due to data-based conditions that are removed or identify false livelocks since all loops in a process without data are always nonterminating.

The main contributions of this paper are a formal rewriting-based symbolic semantics for BPMN with data support and automated verification techniques for checking properties of interest, such as the ones abovementioned. The encod-

ing of BPMN is made using the rewriting logic framework and is fully executable in the Maude system [4]. It comprises BPMN operators such as end/start events, sequence flows, and gateways annotated with data constraints, and support for looping behavior and unbalanced split/join composition (i.e., no systematic correspondence between split and join gateway patterns). Although some important BPMN features, such as events or exceptions, are not considered, the current supported subset is quite expressive.

The symbolic analysis and verification techniques focus on the behavioral aspect of the BPMN processes, which is described as a control-flow graph with data annotations. Data variables and data-based conditions are supported with the help of the recently developed rewriting modulo SMT approach [30], which is well suited to model and analyze reachability properties of infinite-state open systems, exhibiting both internal and external nondeterminism due to the environment. In particular, data variables are logical variables under the control of an SMT-solver and data conditions are constraints over these variables that can be checked for satisfiability with SMT-based decision procedures. This approach ultimately enables the automatic verification of existential reachability properties of BPMN processes with a potential infinite number of initial states. They include deadlock freedom, detection of unreachable states, and reachability of certain states based on data analysis relative to an initial constraint on the data variables.

The Maude specification presented here and several examples are available at http://maude.lcc.uma.es/BPMN-SMT.

The organization of the rest of this paper is as follows. Section 2 introduces BPMN with data and the running example. The example was wrongly designed on purpose, with the intention of identifying its problems through the analysis performed on it later in the paper. Section 3 gives some background about rewriting logic and rewriting modulo SMT. Section 4 introduces the Maude encoding of the considered subset of BPMN, with emphasis on the handling of data. In Sect. 5, analysis techniques for automatically verifying properties on data-aware BPMN processes are presented and illustrated with the help of the running example; it ends with a proposal to correct the running example. Section 6 surveys related work and Sect. 7 concludes the paper.

2 BPMN with Data

BPMN is a workflow-based notation for modeling business processes as collections of tasks that produce specific services or products for particular clients. BPMN is an ISO/IEC standard [15], and can be executed by using different process interpretation engines (e.g., Activiti, Bonita BPM, or jBPM).

In this paper, our goal is not to consider the whole expressiveness of the BPMN language, but to concentrate on its main elements related to control-flow modeling and on data aspects that can be represented in BPMN constructs (variables and conditional flows). By focusing on these aspects, we show how automated analysis is possible for them. Specifically, we consider the node types

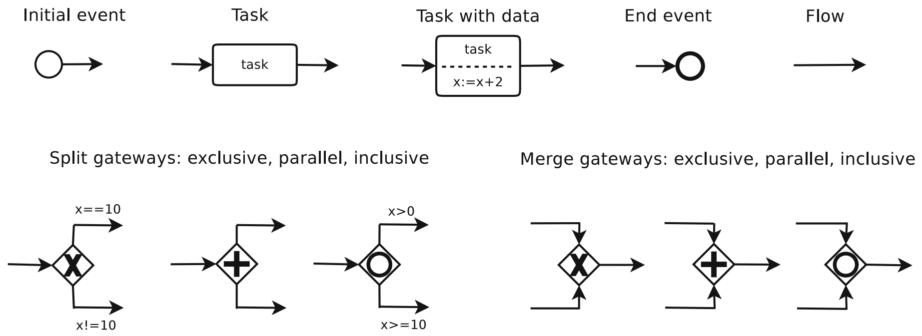


Fig. 1. BPMN syntax augmented with data variables and conditions

event, *task*, and *gateway*, and the edge type *sequence flow*. Figure 1 illustrates the syntax of BPMN supported in this work, including examples of data assignments and conditions at exclusive/inclusive split gateways.

Start and end events are used, respectively, to initialize and terminate processes. A task represents an atomic activity that has exactly one incoming and one outgoing flow. A gateway is used to control the divergence and convergence of the execution flow. A sequence flow describes two nodes executed one after the other, i.e., imposing the execution order.

In BPMN, variables are global to the process. Their initialization and modification is possible at the task level using assignment (:=). Values from the environment can be read using a `lookup` operator. In this paper, we consider basic datatypes (integer, real and Boolean) with usual functions on them. As an example, integers can be manipulated using functions $+$, $-$, $*$, etc. These variables are also used to define conditions in *gateways*.

Gateways are crucial since they are used to model control flow branching and therefore influence the overall process execution. In this paper, we support the three main types of gateways, namely, *exclusive*, *inclusive* and *parallel* gateways. Gateways with one incoming branch and multiple outgoing branches are called *splits*, e.g., inclusive split gateway. Gateways with one outgoing branch and multiple incoming branches are called *merges*, e.g., parallel merge gateway. An exclusive gateway chooses one out of a set of mutually exclusive alternative incoming or outgoing branches depending on the evaluation of conditions on their outgoing flows. For an inclusive gateway, any number of branches among its incoming or outgoing branches may be taken depending on the evaluation of flow conditions. A parallel gateway creates concurrent flows for its outgoing branches or synchronizes concurrent flows for its incoming branches. In this work, we support unbalanced workflows, meaning that each merge gateway does not necessarily have a corresponding split gateway with a correspondence of the branches among outgoing and incoming flows. We also support workflows with looping behaviors.

We assume that BPMN processes are syntactically correct. This can be enforced using existing works, e.g., [11], or using a BPMN engine, as the Activiti or Bonita platforms. Note that any well-typed expression may be used in assignments, using variables, literal and operators, any valid Boolean expression may be used in conditions, and split (resp. merge) gateways may have any positive number of outgoing (resp. incoming) branches.

The semantics of BPMN is described informally in official documents [15,24], and several attempts have been made for giving a formal semantics to subsets of BPMN (see Sect. 6). The semantics of BPMN is usually described using *tokens* representing the evolution of a process execution. A token can enter and leave a task by following the incoming and outgoing flows associated to that task. Tokens are created and consumed at gateways. When a token arrives at a parallel split gateway, it is consumed and one token is generated for every outgoing flow of the split gateway. When a token is consumed at an exclusive split gateway, only one token is created and assigned to the outgoing flow whose condition is evaluated to true. In the case of an inclusive split gateway, when a token is consumed, some new tokens are generated and assigned to the outgoing flows (one for each outgoing flow whose corresponding condition is evaluated to true). We assume that a process starts its execution with exactly one token located at its start event, and finishes its execution when all tokens are at end events. In the following, we will say that a task or flow (or branch) is *active* if it has a token associated with it. E.g., the branches a gateway follows during an execution will be called the active branches.

Running Example. We use as running example an online e-visa service described as a BPMN process shown in Fig. 2. The workflow models the on-line application for a visa, in which the user has to fulfil several forms, provide an electronic copy of her passport and pay the corresponding fees. The process starts with the client initiating the application by filling some basic information. A scanned version of the passport must then be provided. The calculation of the size and quality of the uploaded file is modeled as reading from the environment its size and quality using the `lookup` operator. If the size of the uploaded file does not respect the size limit (2MB), the user may try again to submit another file. Once the file size is within the size limit, the application checks for image quality. Here, again, if the quality is not good enough, the user can upload another scanned version of her passport. The user has up to three attempts to upload the scanned copy of her passport with valid file size and quality. When both the size limit and scan quality respect the imposed thresholds, the request is evaluated and a result is notified to the user (accept or reject). The evaluation is performed by a human agent, which provides her response as an input to the system. In case of acceptance, the user has to pay for fees and the bureau in charge of visa delivery prepares the requested document. Both activities are achieved in parallel because they are independent. Finally, an electronic version of the visa is delivered by email.

In addition to classic BPMN elements, data are used at different places of the process, e.g., for keeping track of the number of attempts or for storing the

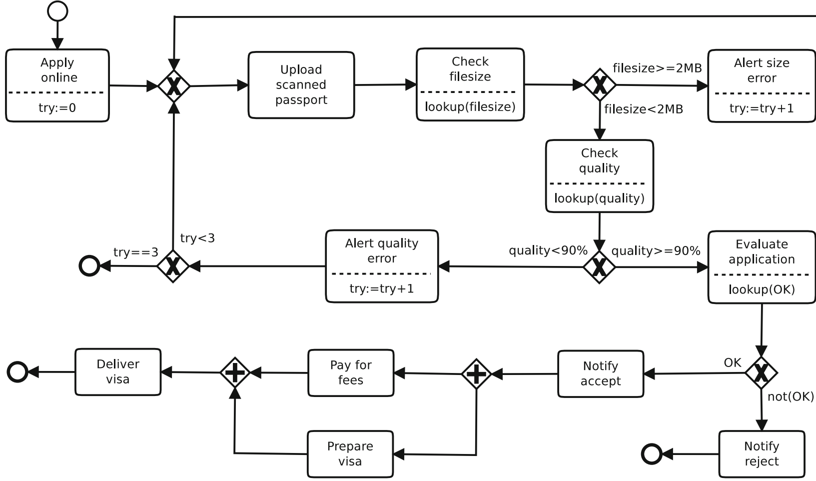


Fig. 2. Running example: e-visa application process

size and quality of the uploaded files. In those usages, the case study includes several interesting data-related features: different variable types (integer, real and Boolean), variable manipulation (assignments, arithmetic expressions and predicates), data-based decisions (depending on input data file size and quality), and external decisions (application evaluation by agent).

As the careful reader has already possibly noticed, and as we will see in Sect. 5, the process in Fig. 2 has several design problems, e.g., the number of attempts for uploading the passport is not correctly handled and checked. The remaining sections of this paper will show how the proposed analysis techniques can detect these issues.

3 Rewriting Logic and Rewriting Modulo SMT in a Nutshell

This section briefly explains order-sorted rewriting logic and rewriting modulo SMT, summarizing Sects. 2, 3, 4 and 5 in [30]. Rewriting logic [20] is a semantic framework that unifies a wide range of models of concurrency. Maude [4] is a language and tool to support the formal specification and analysis of concurrent systems in rewriting logic. Notation on terms, term algebras, and equational theories is used as in, e.g., [1,12].

An *order-sorted signature* Σ is a tuple $\Sigma = (S, \leq, F)$ with a finite poset of sorts $(S, \leq)$ and a set of function symbols F typed with sorts in S. The binary relation $\equiv_{\leq}$ denotes the equivalence relation $(\leq \cup \geq)^{+}$ generated by $\leq$ on S and its point-wise extension to strings in S^{*}. For any sort $s \in S$, the expression $[s]$ denotes the connected component of s, that is, $[s] = [s]_{\equiv_{\leq}}$. A *top sort* in Σ is a sort $s \in S$ such that for all $s' \in [s]$, $s' \leq s$. Let $X = \{X_s\}_{s \in S}$

denote an S-indexed family of disjoint variable sets with each X_s countably infinite. The *set of terms of sort s* and the *set of ground terms of sort s* are denoted, respectively, by $T_\Sigma(X)_s$ and $T_{\Sigma,s}$; similarly, $T_\Sigma(X)$ and T_Σ denote, respectively, the set of terms and the set of ground terms. $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$ denote the corresponding order-sorted Σ-term algebras. 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 and such that $\theta(x) \in T_\Sigma(X)_s$ if $x \in X_s$, for any $x \in X$ and $s \in S$. Substitutions extend homomorphically to terms in the natural way. A substitution θ is called *ground* if and only if $ran(\theta) = \emptyset$. The application of a substitution θ to a term t is denoted by $t\theta$.

An *equational theory* is a tuple (Σ, E), with Σ an order-sorted signature and E a finite collection of (possibly conditional) Σ-equations. An equational theory $\mathcal{E} = (\Sigma, E)$ induces the congruence relation $=_\mathcal{E}$ on $T_\Sigma(X)$ defined for $t, u \in T_\Sigma(X)$ by $t =_\mathcal{E} u$ if and only if $\mathcal{E} \vdash t = u$, where $\mathcal{E} \vdash t = u$ denotes $\mathcal{E}$-provability by the deduction rules for order-sorted equational logic in [21]. The expressions $\mathcal{T}_\mathcal{E}(X)$ 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 *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. $\mathcal{R} = (\Sigma, E, R)$ is called a *topmost rewrite theory* if it has a top sort *Conf* such that no operator in Σ has *Conf* as argument sort and each rule $l \to r$ **if** $\phi \in R$ satisfies $l, r \in T_\Sigma(X)_{Conf}$ and $l \notin X$. A rewrite theory $\mathcal{R}$ induces a rewrite relation $\to_\mathcal{R}$ on $T_\Sigma(X)$ defined for every $t, u \in T_\Sigma(X)$ by $t \to_\mathcal{R} u$ if and only if there is a rule $(l \to r$ **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 tuple $\mathcal{T}_\mathcal{R} = (\mathcal{T}_{\Sigma/E}, \to_\mathcal{R}^*)$ is called the *initial reachability model of* $\mathcal{R}$ [2].

Appropriate requirements are needed to make an equational theory $\mathcal{E}$ *admissible*, i.e., *executable* in rewriting languages such as Maude [4]. 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 regular and linear 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. Furthermore, it is assumed that the equations E can be oriented into a set of (possibly conditional) sort-decreasing, operationally terminating, confluent rewrite rules $\overrightarrow{E}$ modulo B. The rewrite system $\overrightarrow{E}$ is *sort decreasing* modulo B if and only if for each $(t \to u$ **if** $\gamma) \in \overrightarrow{E}$ and substitution θ, $ls(t\theta) \geq ls(u\theta)$ if $(\Sigma, B, \overrightarrow{E}) \vdash \gamma\theta$. The system $\overrightarrow{E}$ is *operationally terminating* modulo B [9] if and only if there is no infinite well-formed proof tree in $(\Sigma, B, \overrightarrow{E})$. Furthermore, $\overrightarrow{E}$ is *confluent* modulo B if and only if for all $t, t_1, t_2 \in T_\Sigma(X)$, if $t \to_{E/B}^* t_1$ and $t \to_{E/B}^* t_2$, then there is $u \in T_\Sigma(X)$ such that $t_1 \to_{E/B}^* u$ and $t_2 \to_{E/B}^* u$. The term $t{\downarrow_{E/B}} \in T_\Sigma(X)$ denotes the *E-canonical form* of t modulo B so that $t \to_{E/B}^* t{\downarrow_{E/B}}$ and $t{\downarrow_{E/B}}$ cannot be further reduced by $\to_{E/B}$. Under sort-decreasingness, operational termination, and confluence, the term $t{\downarrow_{E/B}}$ is unique up to B-equality. For a rewrite theory $\mathcal{R}$, the rewrite relation $\to_\mathcal{R}$ is undecidable in general, even if its underlying equational theory is admissible,

unless conditions such as coherence [31] are given (i.e., whenever rewriting with $\rightarrow_{R/E \cup B}$ can be decomposed into rewriting with $\rightarrow_{E/B}$ and $\rightarrow_{R/B}$).

A rewrite theory $\mathcal{R} = (\Sigma, E \uplus B, R)$ modulo a built-in subtheory $\mathcal{E}_0 = (\Sigma_0, E_0 \uplus B_0) \subseteq (\Sigma, E \uplus B)$ is a topmost rewrite theory with a signature of built-ins $\Sigma_0 = (S_0, \leq_0, F_0)$. The *ground rewrite relation* $\rightarrow_{\mathcal{R}}$ induced by a rewrite theory with built-ins $\mathcal{R}$ is the topmost rewrite relation defined for any $t, u \in T_{\Sigma,Conf}$ as follows: $t \rightarrow_{\mathcal{R}} u$ if and only if there is a rule $l \rightarrow r$ **if** ϕ in R and a ground substitution $\sigma : X \longrightarrow T_{\Sigma}$ such that $t =_{E \uplus B} l\sigma$, $u =_{E \uplus B} r\sigma$, and $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$.

The symbolic rewrite relation induced by a rewrite theory with built-ins $\mathcal{R}$ operates over pairs $(t\,; \varphi)$, called *constrained terms*, where $t \in T_{\Sigma}(X_0)_{Conf}$ is a term, $\varphi \in QF_{\Sigma_0}X_0$ is a constraint of built-ins, and $X_0 \subseteq X$ are the variables with sorts in S_0. Each formula in $QF_{\Sigma_0}X_0$ is a Boolean combination of atoms, where an atom is a Σ_0-equation with variables in X_0. For any term $t \in T_{\Sigma}(X)_{Conf}$ and constraint φ, the *denotation* $[\![t]\!]_{\varphi}$ is defined as $[\![t]\!]_{\varphi} = \{t' {\in} T_{\Sigma,Conf} \mid (\exists \sigma : X {\longrightarrow} T_{\Sigma})\ t' {=_E} t\sigma \wedge \mathcal{T}_{\mathcal{E}_0} \models \varphi\sigma\}$. Given a rewrite rule $((l\,; \phi_l) \rightarrow (r\,; \phi_r)$ **if** $\phi) \in R$, with $l, r \in T_{\Sigma}(X)_{Conf}$ and $\phi \in QF_{\Sigma_0}X_0$, a constrained term $(t\,; \phi_t) \in T_{\Sigma}(X_0)_{Conf} \times QF_{\Sigma_0}X_0$ symbolically rewrites to a constrained term $(u\,; \phi_u) \in T_{\Sigma}(X_0)_{Conf} \times QF_{\Sigma_0}X_0$ (denoted by $(t\,; \phi_t) \rightsquigarrow_{\mathcal{R}} (u\,; \phi_u)$) if and only if there is a substitution θ such that: (a) $l\theta =_{E \uplus B} t$ and $r\theta =_{E \uplus B} u$, (b) $\mathcal{T}_{\Sigma/E \uplus B} \models (\phi_l \wedge \phi_t\theta) \Leftrightarrow \phi_u$, and (c) ϕ_u is $\mathcal{T}_{\Sigma/E \uplus B}$-satisfiable. Condition (a) can be solved by matching as in the definition of $\rightarrow_{\mathcal{R}}$ above. Condition (b) can be met by setting ϕ_u to be $\phi_l \wedge \phi_t\theta$. Condition (c) is checked with the help of decision procedures available from an SMT solver via the function *check-sat*. Observe that, up to the choice of the semantically equivalent φ_u, the symbolic relation $\rightsquigarrow_{\mathcal{R}}$ is deterministic in the sense of being determined by the rule and the substitution θ (here it is assumed that variables in the rules are disjoint from the ones in the target terms). The reader is referred to [30] for details about rewriting modulo SMT and the correspondence between $\rightarrow_{\mathcal{R}}$ and $\rightsquigarrow_{\mathcal{R}}$.

4 Symbolic Specification

The symbolic semantics of BPMN is defined as a rewrite theory $\mathcal{R}$, with built-ins $\mathcal{E}_0$ and topsort `Conf`. A symbolic state is a constrained term $(t\,; \varphi)$, where t is a configuration of objects and φ a constraint. The objects in t represent the set of nodes and flows of a process, and keep information of the execution of the process. The constraint φ ranges over the built-in sorts `Boolean`, `Integer`, and `Real`, of Booleans, integers, and reals, respectively, and it maintains the bookkeeping of constraints accumulated during execution. For example, a constraint φ can encode the possible values for an integer variable that are possible after the process is executed.

Figure 3 includes a representation in Maude of the e-visa application process as a `Process` object. This object represents the static part of the process, and identifies its nodes (attribute `nodes`) and flows (attribute `flows`). The process `pid` is parametric on 4 built-in variables, namely, `OK` (of sort `Boolean`), `try` and

filesize (of sort Integer), and quality (of sort Real). Thus the constraints accumulated during the execution of this process range over these 4 variables.

```
< pid : Process |
  nodes :
    (start(initial, sf1),
     end(final1, sf25),
     end(final2, sf24),
     end(final3, sf14),
     task(t1, "Apply online", sf1, sf2, try := 0),
     task(t2, "Upload scanned passport", sf3, sf4),
     task(t3, "Check filesize", sf4, sf5, lookup(filesize)),
     task(t4, "Alert size error", sf6, sf8, try := try + 1),
     task(t5, "Check quality", sf7, sf9, lookup(quality)),
     task(t6, "Evaluate application", sf11, sf15, lookup(OK)),
     task(t7, "Alert quality error", sf10, sf12, try := try + 1),
     task(t8, "Notify accept", sf16, sf18),
     task(t9, "Pay for fees", sf19, sf21),
     task(t10, "Deliver visa", sf23, sf24),
     task(t11, "Notify reject", sf17, sf25),
     task(t12, "Prepare visa", sf20, sf22),
     merge(m1, exclusive, (sf2, sf8, sf13), sf3),
     merge(m2, parallel, (sf21, sf22), sf23),
     split(s1, exclusive, sf5, (sf6, filesize >= 2) (sf7, filesize < 2)),
     split(s2, exclusive, sf9, (sf10, quality < 9/10) (sf11, quality >= 9/10)),
     split(s3, exclusive, sf15, (sf17, OK === false) (sf16, OK === true)),
     split(s4, parallel, sf18, (sf19, sf20)),
     split(s5, exclusive, sf12, (sf13, try < 3) (sf14, try === 3)))
  flows :
    (flow(sf1), flow(sf2), ..., flow(sf24), flow(sf25)) >
```

Fig. 3. Running example: representation in Maude of the e-visa application process

The initial node of the process is represented as the term start(initial, sf1) specifying that the initial node has outgoing flow sf1. Task nodes are specified by an identifier, a label, an input flow, an output flow, and a list of assignments. An assignment has the form x := E, where x is a variable, and E an expression over the variables with the same sort of x. In this particular example, x is a variable in the set {OK, try, filesize, quality} and E an expression over the same set of variables. For instance, the assignment try := try + 1 in task t4 indicates that the value represented by the variable try is to be incremented in one unit. For the reading of external variables we use the lookup operator. E.g., variables filesize and quality are read in tasks t3 and t5, respectively. Split nodes are specified by an identifier, a type (exclusive, inclusive, or parallel), an incoming flow, and a nonempty list of outgoing flows. Exclusive and inclusive gateways are equipped with constraints associated to the outgoing flows: these represent conditions over the data of the process. For instance, in the exclusive split gateway s2, the list

```
(sf10, quality < 9/10) (sf11, quality >= 9/10)
```

specifies that the flow sf10 is triggered when the quality of the picture is below 90% and that the flow sf11 is taken otherwise. Merge nodes are specified by an identifier, a type (the same types of split gateways), a set of incoming flows, and

```
< sim : Simulation |
    tokens : token(sf1),
    constr : true,
    varidx : (v(OK) |-> 0, v(try) |-> 0, v(filesize) |-> 0, v(quality) |-> 0) >
```

Fig. 4. Running example: simulation in Maude of the e-visa application process

an outgoing flow. End nodes are specified by an identifier and an incoming flow. Flows are specified just by an identifier.

Figure 4 shows a representation in Maude of an execution state of the process as the object `sim`. This object gathers the dynamic information used along execution, and has attributes identifying the set of tokens (attribute `tokens`), the constraint accumulated during execution (attribute `constr`), and a map (of sort `Map{SymbVar, Nat}`) for indexing the built-in variables present in the state (attribute `varidx`). In this example, the token to be processed next is at flow `sf1`, the constraint is `true` (i.e., the empty one), and all variables in the process are indexed at 0.

The index of a variable can increase during execution, which is key because an expression over the built-ins is evaluated with respect to a given variable indexing. Since the variables in a process are "mathematical" variables, they need to be treated with a predicate-transformer approach (opposed to the imperative programming-like approach). For example, given the variable indexing `v(try) |-> 4`, the expression `try + 1` is interpreted as `try#4 + 1`, meaning that the interpretation of the increment of `try` by one unit is to be done with respect to the 'latest' version of the variable. When an assignment such as `try := try + 1` is evaluated, it creates a constraint that is added to the global constraint and depends on the given variable indexing. For example, with the variable indexing `v(try) |-> 4`, the evaluation of `try := try + 1` results in the constraint `try#5 === try#4 + 1`, meaning that the "new" value of `try` corresponds to its previous value incremented by one. The `lookup` operator works similarly, although in this case there is no restriction on the new value.

The concurrent transitions of $\mathcal{R}$ are specified by rewrite rules that update the simulation object `sim`. Specifically, there are 12 rewrite rules that model the different actions that may happen in the system, e.g., start or end the execution of a process, handle the arrival of a token to split and merge gateways, tasks, and flows. For illustration purposes, we explain in the rest of this section two of these rules.

Figure 5 shows the rewrite rule that handles execution of a task. Intuitively, executing a task results in a new symbolic state in which the constraint is updated by accumulating the constraints that result from the list of assignments associated to the task, if possible. Since handling an assignment changes the variable indexing kept in the object `sim`, a new version of the variable indexing term needs to be computed. Both the constraint resulting from a list of assignments and the new variable indexing are computed with the help of the auxiliary

```
crl [execTask] :
    < PId : Process |
          nodes : (task(NId, TaskName, FId1, FId2, AsgL), Nodes),
          flows : (flow(FId2), Flows),
          Atts >
    < SId : Simulation |
          tokens : (token(NId), Tks),
          constr : B,
          varidx : VMp,
          Atts1 >
 => < PId : Process |
          nodes : (task(NId, TaskName, FId1, FId2, AsgL), Nodes),
          flows : (flow(FId2), Flows),
          Atts >
    < SId : Simulation |
          tokens : (token(FId2), Tks),
          constr : (B and B1),
          varidx : VMp1,
          Atts1 >
 if (VMp1, B1) := prepare-update(AsgL, VMp) .
```

Fig. 5. Execution of a task with a list of assignments

function `prepare-update`. For example, the evaluation of the term

```
prepare-update((X := X + 1) (Y := Y + X), (v(X) |-> 4, v(Y) |-> 0))
```

returns the pair

```
((v(X) |-> 5, v(Y) |-> 1), X#5 === X#4 + 1 and Y#1 === Y#0 + X#5)
```

where the index of both variables `X` and `Y` is incremented, and the constraint chains the sequential assignment performed on these variables. Note that this rule does not check the satisfiability of the constraint `B1` resulting from the assignment in conjunction with the constraint `B` from the state; this is because if the latter is satisfiable, so is their conjunction since the newly constrained variables in `B1` are fresh with respect to `B`.

One interesting case for symbolic specification can be observed in the semantics of inclusive gateways, axiomatized by the rule in Fig. 6. In a version without data, when a token arrives at a split gateway, any number of outgoing flows are selected and assigned newly created tokens that start executing right away. However, in a symbolic version with data, the situation is more complex because *all* possible flow selections need to be considered and checked for satisfiability relative to the global constraint. E.g, if there are n outgoing flows, then there are at most $2^n - 1$ possible selections satisfiable relative to the global constraint. Each one of these scenarios, computed by the auxiliary function `gen`, can be identified by the set of the m constraints $(0 < m \leq n)$ associated to the selected outgoing flows, say Γ, and the set of the $n - m$ constraints associated to the outgoing flows that are not selected, say Δ. Then, the constraint `B1` associated to such selection is logically equivalent to the following conjunction of constraints:

$$\bigwedge_{\gamma \in \Gamma} \gamma \wedge \bigwedge_{\delta \in \Delta} \neg\delta.$$

Such a constraint needs to be interpreted with respect to the current variable indexing, resulting in a new constraint, say `B2`; this is done with the help of the auxiliary function `process-expression`. Finally, a split is possible if the constraint representing the symbolic split is compatible with the constraint `B` accumulated in the state, i.e., if `check-sat(B and B2)` evaluates to true.

5 Symbolic Execution and Reachability Analysis

Symbolic execution and reachability analysis can be useful for exploring infinitely many system executions at once. In the rewriting modulo SMT implementation available from Maude, symbolic execution of the rewrite theory $\mathcal{R}$ presented in Sect. 4 uses a combination of term rewriting, matching modulo axioms, and SMT solving. This consists in applying equations and rewrite rules from an initial term (e.g., from an initial state such as the one in Fig. 3) and querying the SMT solver for checking satisfiability of constraints at each rewrite step, when necessary. The full potential of rewriting modulo SMT can be exploited for solving existential reachability queries in the initial model $\mathcal{T}_\mathcal{R}$ of a rewrite theory $\mathcal{R}$ modulo built-ins $\mathcal{E}_0$. The type of *existential reachability* question that rewriting modulo SMT can answer can be formulated as follows:

are there states in $[\![t]\!]_\phi$ *that can reach a state in* $[\![u]\!]_\psi$*?*

This question is especially useful for symbolically proving or disproving safety properties, such as inductive invariants and deadlock freedom of $\mathcal{T}_\mathcal{R}$: when $[\![u]\!]_\psi$ is a set of *bad* states, the goal is to check whether reaching a state in $[\![u]\!]_\psi$ is possible.

```
crl [splitGatewayInclusive] :
    < PId : Process |
         nodes : (split(NId, inclusive, FId, LTIB), Nodes),
         flows : Flows,
         Atts >
    < SId : Simulation |
         tokens : (token(FId), Tks),
         constr : B,
         varidx : VMp,
         Atts1 >
 => < PId : Process |
         nodes : (split(NId, inclusive, FId, LTIB), Nodes),
         flows : Flows,
         Atts >
    < SId : Simulation |
         tokens : (tokens(IL), Tks),
         constr : (B and B2),
         varidx : VMp,
         Atts1 >
 if ILL1 . IL . ILL2 := gen(project1(LTIB))
 /\ B1 := get-constr(IL, LTIB)
 /\ B2 := process-expression(VMp, B1)
 /\ check-sat(B and B2) .
```

Fig. 6. Arrival of a token to an inclusive split gateway

In the rest of this section, we show how to use symbolic rewriting and symbolic reachability analysis for verifying properties of interest on data-aware processes encoded into Maude's rewriting logic. This is illustrated using the e-visa application process introduced in Sect. 2. Note that, beyond toy examples and the e-visa application process we use as running example in this paper, we have also applied our approach to two other examples we found in the literature: a drug store process [14] and a DMN process on application file handling [25, Sect. 11.2].

5.1 Symbolic Rewriting

As for regular rewrite theories, rewriting may be useful for gathering a first understanding of the whereabouts of our systems. Consider the running example in Fig. 2. For the initial state corresponding to the terms in Figs. 3 and 4, the following rewrite command in Maude symbolically executes the process for ten consecutive rewrite steps:

```
rew [10] initSystem(v(OK:Boolean) v(TRY:Integer) v(FSIZE:Integer) v(QUAL:Real)) .
```

The `initSystem` operator generates the initial state depending on the specified list of variables. Maude's output to this command corresponds to one of the possible states reached after ten rewrite steps; in this case, the output is the following term:

```
< p : Process | nodes : ..., flows : ... >
< s : Simulation |
    tokens : token(t2),
    constr : (TRY#1:Integer === 0 and
              FSIZE#1:Integer >= 2 and
              TRY#2:Integer === TRY#1:Integer + 1,
    varidx : (v(OK) |-> 0, v(TRY) |-> 2, v(FSIZE) |-> 1, v(QUAL) |-> 0) >
```

In this state, there is exactly one token, at task `t2`, the "Upload scanned passport" task. The constraint indicates that the `TRY` variable has value 1 and that `FSIZE#1` is at least 2. After a first upload of an oversized file, task "Alert size error" was executed, where the `TRY` variable got increased, and then moved to the task `t2`. Note that the index of variable `TRY` is set to 2 because of the initial assignment on this variable in the "Apply online" task, and its update in "Alert size error". The index for `FSIZE` is 1.

A constraint like the one in this final state can also be interpreted as the conditions on the initial state and interactions to lead us to such state. Although a symbolic state may represent an infinite number of concrete states, as a symbolic path may represent an infinite number of concrete executions, in this case, we can say that any execution in which we upload an oversized file will lead to this state in ten rewriting steps.

5.2 Symbolic Reachability Analysis

Beyond symbolic rewriting, symbolic reachability is useful in order to answer a number of interesting questions such as the following ones:

– What are the reachable states after execution of n rewrite steps? Notice that each rewrite represent an infinite number of possible executions.
– Is it possible for an input variable to be assigned a certain value? As an example, can we check whether our running example can reach a state in which the TRY variable takes a value greater than 3?
– Are there deadlocks in the process?
– Does a process have unreachable flows or tasks?

In the rest of this section, we will see how reachability analysis can be used to answer these questions. Consider the following existential query, where $\mathcal{R}$ is the rewrite theory presented in Sect. 4 corresponding to the running example in Fig. 2:

$$\mathcal{T}_{\mathcal{R}} \models (\exists \texttt{OK:Boolean}, \texttt{TRY:Integer}, \texttt{FSIZE:Integer}, \texttt{QUAL:Real}, \texttt{St:Conf})$$
$$\texttt{initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL))} \overset{*}{\rightarrow}_{\mathcal{R}} \texttt{St}$$
$$\wedge \text{ ``}\texttt{St} \text{ is } \rightarrow_{\mathcal{R}} -\text{irreducible''} \wedge \text{ ``}\texttt{TRY = 3} \text{ in } \texttt{St}\text{''}.$$

This query asks whether we can reach from our initial state an irreducible state (w.r.t. the rewrite relation) in which the value of the TRY variable can be 3. Observe that infinitely many states need to be considered in the query because some of the variables range over infinite domains (e.g., integer and real numbers). This means that such a query, in general, is beyond the reach of ground rewriting and would require, for instance, inductive techniques over the rewrite relation. However, the following search command can be issued in Maude to find a proof (or a counterexample) of the reachability query for $\rightarrow_{\mathcal{R}}$ using the symbolic rewrite relation $\rightsquigarrow_{\mathcal{R}}$:

```
search [1] initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL)) =>! St
  such that
    check-sat(get-constr(St) and process-expression(get-varidx(St), try === 3)) .
```

Note the use of `[1]` and `=>!` to indicate, respectively, our interest in finding exactly one witness to the existential reachability query and that such a witness needs to be irreducible (i.e., no rule can be applied to it). The function calls `get-constr(St)` and `get-varidx(St)` return, respectively, the constraint and the variable indexing from any execution state `St`. The satisfiability of the constraint `TRY === 3` needs to be checked against the variable indexing at each corresponding execution state; this is achieved by invoking the auxiliary function `process-expression`. The above search command generates the following output:

```
Solution 1 (state 155)
states: 171  rewrites: 2111 in 68ms cpu (70ms real) (30874 rewrites/second)
St --> < p : Process | nodes : ..., flows : ... >
       < s : Simulation |
           tokens : empty,
           constr : (TRY#1:Integer === 0 and
                     FSIZE#1:Integer >= 2 and
                     TRY#2:Integer === TRY#1:Integer + 1 and
                     FSIZE#2:Integer >= 2 and
                     TRY#3:Integer === TRY#2:Integer + 1 and
                     FSIZE#3:Integer < 2 and
                     QUAL#1:Real < 9/10 and
                     TRY#4:Integer === TRY#3:Integer + 1 and
```

```
                  TRY#4:Integer === 3),
         varidx : (v(OK) |-> 0, v(TRY) |-> 4, v(FSIZE) |-> 3, v(QUAL) |-> 1) >
```

This means that there is at least one ground execution from an initial state that reaches an irreducible state where the value of the TRY variable is 3. Indeed, the constraint tells us how to reach such a state: three files must be uploaded, the first two with size over 2 MB and the third with a quality under the threshold. Observe that this solution is actually a final state because all tokens have been processed. Note also the ellipses for brevity. The process does not change along the execution. The indication of requesting only one solution is important. Without it, Maude would have kept giving more and more solutions, in which any number of oversized passport files are uploaded due to the unguarded loop. This clearly points out a design error. A corrected version of the process is given below.

As mentioned earlier, the symbolic specification $\mathcal{R}$ presented in Sect. 4 can be used to automatically check for other safety properties such as deadlock freedom. In general, a reachability query associated to having a deadlock in $\mathcal{T}_{\mathcal{R}}$ can be cast as the following satisfaction relation:

$$\mathcal{T}_{\mathcal{R}} \models (\exists \overrightarrow{x}, \texttt{St:Conf})\, \texttt{initSystem(}\overrightarrow{x}\texttt{)} \overset{*}{\rightarrow}_{\mathcal{R}} \texttt{St} \wedge \\ \text{“}\texttt{St} \text{ is } \rightarrow_{\mathcal{R}} \text{–irreducible”} \wedge \\ \text{“}\texttt{St} \text{ is not final”}.$$

In this formula, $\overrightarrow{x}$ denotes the list of variables input to the process specified by $\mathcal{R}$. The condition on the irreducibility of the state St is the same one appearing in the previous reachability goal. In the symbolic semantics $\mathcal{R}$, a state is considered final whenever all tokens in the state have been consumed. This is of practical importance because checking for final states can be decided by checking the contents of the token set, namely, by checking if the set is empty. The following search command in Maude can be used to find deadlocks in the running example process:

```
Maude> search [1] in RUN :
         initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL)) =>! St
           such that get-tokens(St) =/= empty .
Solution 1 (state 249)
states: 265  rewrites: 2852 in 97ms cpu (99ms real) (29183 rewrites/second)
St --> < p : Process | nodes : ..., flows : ... >
       < s : Simulation |
         tokens : token(sf12, 0),
         constr : (TRY#1:Integer === 0 and
                    FSIZE#1:Integer >= 2 and
                    TRY#2:Integer === TRY#1:Integer + 1 and
                    FSIZE#2:Integer >= 2 and
                    TRY#3:Integer === TRY#2:Integer + 1 and
                    FSIZE#3:Integer >= 2 and
                    TRY#4:Integer === TRY#3:Integer + 1 and
                    FSIZE#4:Integer < 2 and
                    QUAL#1:Real < 9/10 and
                    TRY#5:Integer === TRY#4:Integer + 1),
         varidx : (v(OK)|-> 0, v(TRY)|-> 5, v(FSIZE)|-> 4, v(QUAL)|-> 1) >
```

The witness provided shows that a deadlock can actually be reached. The constraint in the deadlock state tells us that by uploading a file with poor quality after three oversized files, we reach a state in which there is a token at the flow `sf12`, the outgoing flow of task "Alert quality error". Note that in that state the variable `TRY` has value 4, and none of the alternative branches of the following exclusive split can be triggered.

Let us now illustrate the check of whether certain flows or tasks are reachable. The following command proves that the task "Pay for fees" cannot be reached with an oversized file.

```
search [1,500]
  initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL))
  =>! St
  such that token-at("Pay for fees", St)
            and
            check-sat(
              get-constr(St)
              and
              process-expression(get-varidx(St), gen-intvar("FSIZE") > 2)) .
No solution.
states: 8846
rewrites: 117728 in 7818ms cpu (8151ms real) (15057 rewrites/second)
```

In this section we have included the results provided by Maude on the number of executions and time obtained when executing all the rewrite and search commands. If we look at the last search command above, we can see that the exploration of 500 states took around 8 seconds. This number is rather low for Maude, if compared to standard rewriting/search. Notice however that symbolic rewriting/search involves additional satisfiability checks during the rewriting process, which are handled by invocation to back end SMT solvers.

Last but not least, let us give a corrected version of the running example (Fig. 7) where the problem coming from an erroneous handling of the number of attempts has been resolved. This was achieved using an exclusive split gateway before the scanned passport upload, which checks for the number of attempts. If this number is greater than three, the process terminates. Note that all the properties mentioned beforehand in this section are satisfied by this new version of the process as we will show in the final part of this section.

There are now only eight possible ways to reach a final state with the variable `TRY` with value 3, showing the different combinations for uploading an invalid file at most 3 times:

```
search initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL))
       =>! St
       such that check-sat(get-constr(St) and
                 process-expression(get-varidx(St), gen-intvar("TRY") === 3)) .

Solution 1 (state 146)
states: 158  rewrites: 2042 in 62ms cpu (64ms real) (32520 rewrites/second)
St --> < p : Process | nodes : ..., flows : ... >
       < s : Simulation |
         tokens : empty,
         gtime : 0,
         constr : (TRY#1:Integer === 0 and
                   TRY#1:Integer < 3 and
                   FSIZE#1:Integer >= 2 and
```

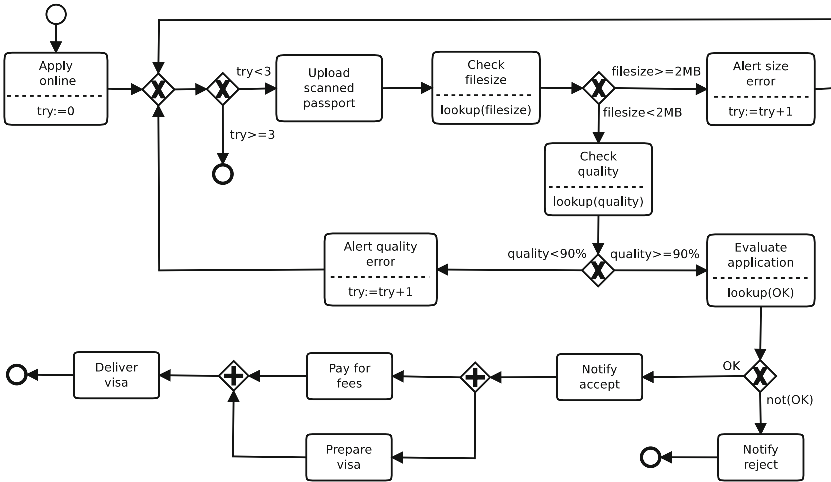


Fig. 7. Running example: e-visa application process (V2)

```
                   TRY#2:Integer === TRY#1:Integer + 1 and
                   TRY#2:Integer < 3 and
                   FSIZE#2:Integer >= 2 and
                   TRY#3:Integer === TRY#2:Integer + 1 and
                   TRY#3:Integer < 3 and
                   FSIZE#3:Integer >= 2 and
                   TRY#4:Integer === TRY#3:Integer + 1 and
                   TRY#4:Integer >= 3),
         varidx : (v(OK) |-> 0, v(TRY) |-> 4, v(FSIZE) |-> 3, v(QUAL) |-> 0) >

...

Solution 8 (state 259)
states: 267  rewrites: 3457 in 134ms cpu (136ms real) (25739 rewrites/second)
St --> < p : Process | nodes : ..., flows : ... >
       < s : Simulation |
         tokens : empty,
         gtime : 0,
         constr : (TRY#1:Integer === 0 and
                   TRY#1:Integer < 3 and
                   FSIZE#1:Integer < 2 and
                   QUAL#1:Real < (9/10).Real and
                   TRY#2:Integer === TRY#1:Integer + 1 and
                   TRY#2:Integer < 3 and
                   FSIZE#2:Integer < 2 and
                   QUAL#2:Real < (9/10).Real and
                   TRY#3:Integer === TRY#2:Integer + 1 and
                   TRY#3:Integer < 3 and
                   FSIZE#3:Integer < 2 and
                   QUAL#3:Real < (9/10).Real and
                   TRY#4:Integer === TRY#3:Integer + 1 and
                   TRY#4:Integer >= 3),
         varidx : (v(OK) |-> 0, v(TRY) |-> 4, v(FSIZE) |-> 3, v(QUAL) |-> 3) >
```

The new version of the process is deadlock free:

```
search [1,500]
  initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL))
  =>! St
  such that get-tokens(St) =/= empty .
```

```
No solution.
states: 293  rewrites: 3333 in 123ms cpu (126ms real) (26882 rewrites/second)
```

Finally, as for the previous version, there is no way to reach the `"Pay for fees"` task (`t9`) with an oversized file:

```
search [1,500] initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL))
       =>! St
       such that token-at("Pay for fees", St)
                 and
                 check-sat(
                   get-constr(St)
                   and
                   process-expression(get-varidx(St), gen-intvar("FSIZE") > 2)) .

No solution.
states: 293  rewrites: 3530 in 133ms cpu (135ms real) (26518 rewrites/second)
```

```
search [1,500] initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL))
       =>! St
       such that token-at(t9, St)
                 and
                 check-sat(
                   get-constr(St)
                   and
                   process-expression(get-varidx(St), gen-intvar("FSIZE") > 2)) .

No solution.
states: 293  rewrites: 3490 in 132ms cpu (134ms real) (26390 rewrites/second)
```

6 Related Work

Several works have focused on providing rigorous definitions and formal semantics for business processes using Petri nets, process algebras, or abstract state machines, see, *e.g.*, [6,7,13,17–19,25,28,32,33]. The main differences with respect to these related works are our focus on data-aware workflow models and the fact that our encoding gives a symbolic semantics to BPMN by translation to Maude.

As far as data-based analysis is concerned, Decision Model and Notation (DMN) is a recent OMG standard for modeling decisions in an interchangeable format. DMN can be used into workflow-based notations for representing conditions. [3] proposes a formal semantics of DMN decision tables, a notion of DMN table correctness, and algorithms that check the detection of overlapping rules and missing rules. These algorithms have been implemented in the DMN toolkit and validated through empirical evaluation. Our modeling language provides the same expressiveness as decision tables existing in DMN, but our analysis techniques go further since they allow to verify properties of interest on the whole flow of control taking data and conditions into account.

Herbert and Sharp [14] choose to simplify the modeling of exclusive/inclusive split gateways by considering probabilities instead of conditions. They propose an algorithm for translating a BPMN subset extended with probabilistic information into the guarded command language used by PRISM. This enables model

checking of quantitative properties of business processes such as transient probabilities, occurrence of events, and best-/worst-case scenarios. [14] uses the notion of *rewards*, from Markov Models [31], as annotated values that can be used to keep track of quantities of interest (e.g., execution times, number of iterations, etc.) in processes.

In [27], Prandi et al. propose a formalization for BPMN models which supports rewards and probabilistic elements. They propose a conversion of BPMN models into a model expressed in the Calculus for Orchestration of Web Services (COWS) [28], which can then be analyzed by using model checking.

[22] focuses on the analysis of choreography models. The main property of interest in that context is called *conformance* and aims at checking whether the distributed implementation and the choreography behave identically. The authors mainly focus on data description. Their approach supports choreographies extended with conditions and relies on SMT solving for conformance checking.

Several authors have used rewriting logic and Maude to model and analyze BPMN processes. El-Saber and Boronat [11] propose a translation of BPMN into rewriting logic with a special focus on data-based decision gateways. They provide mechanisms to avoid structural issues in workflows such as flow divergence by introducing the notion of "well-formed" BPMN process. Their approach aims at avoiding incorrect syntactic patterns whereas we propose automated analysis at the semantic level. Kheldoun et al. [16] propose high-level Petri nets and to use Maude's LTL model checker for, respectively, specifying BPMN processes and analyzing behavioral properties. They also focus on handling exceptions and activity cancellation. Durán and Salaün used Maude to represent BPMN processes enriched with time features in [10]. In this paper, they show how real-time analysis of such BPMN processes could be performed. Specifically, they used simulations, reachability analysis and model checking, and calculate certain properties such as minimum and maximum expected response times, maximum degree of parallelism, and synchronization times. Corradini et al. present in [5] their tool BProVe, a friendly tool for the verification of business processes modeled in BPMN. The tool accepts BPMN processes in standard notation and can perform checks of soundness and safeness on them, as defined in [34] and [8], respectively, using Maude's LTL model checker.

7 Concluding Remarks

In this paper, we have focused on the BPMN modelling language enhanced with some constructs for supporting the description of data (variables and conditions). We have proposed a symbolic semantics for BPMN with data using Maude's rewriting logic framework. The transformation to Maude allows us to verify properties on the process models such as deadlock freedom or reachability of certain states based on data analysis. These verifications are automated relying on rewriting modulo SMT techniques. We have applied our approach to several use cases, such as an online e-visa application.

As far as future work is concerned, a first perspective is to enlarge the number of properties we can verify. A first step would be to be able to also detect the absence of livelocks, and in general to be able to use model checking on these specifications. Using the techniques in [30] we could already use model checking, however we need to work on mechanisms for dealing with infinite state spaces. We are currently considering equational abstractions, but some formal issues need to be solved. A second perspective aims at combining our recent work on BPMN with time [10] and the data support presented in this paper. This would result in a richer language for modelling processes as well as verification techniques working on data and time aspects at the same time.

Acknowledgments. The authors would like to thank the anonymous reviewers for their valuable comments on an earlier draft of this paper. F. Durán has been partially supported by Spanish MINECO/FEDER project TIN2014-52034-R and Univ. Málaga, Campus de Excelencia Internacional Andalucía Tech. The work of C. Rocha was partially supported by CAPES, Colciencias, and INRIA via the STIC AmSud project "EPIC: EPistemic Interactive Concurrency" (Proc. No 88881.117603/2016-01), and by Capital Semilla 2017, project "SCORES: Stochastic Concurrency in Rewrite-based Probabilistic Models" (Proj. No. 020100610).

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1999)
2. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theor. Comput. Sci. **360**(1–3), 386–414 (2006)
3. Calvanese, D., Dumas, M., Laurson, Ü., Maggi, F.M., Montali, M., Teinemaa, I.: Semantics and analysis of DMN decision tables. In: La Rosa, M., Loos, P., Pastor, O. (eds.) BPM 2016. LNCS, vol. 9850, pp. 217–233. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45348-4_13
4. Lincoln, P., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
5. Corradini, F., Fornari, F., Polini, A., Re, B., Tiezzi, F., Vandin, A.: BProVe: a formal verification framework for business process models. In: Proceedings of ASE, pp. 217–228. IEEE Computer Society (2017)
6. Decker, G., Weske, M.: Interaction-centric modeling of process choreographies. Inf. Syst. **36**(2), 292–312 (2011)
7. Dijkman, R., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. **50**(12), 1281–1294 (2008)
8. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. **50**(12), 1281–1294 (2008)
9. Durán, F., Lucas, S., Marché, C., Meseguer, J., Urbain, X.: Proving operational termination of membership equational programs. High. Order Symb. Comput. **21**(1–2), 59–88 (2008)
10. Durán, F., Salaün, G.: Verifying timed BPMN processes using Maude. In: Jacquet, J.-M., Massink, M. (eds.) COORDINATION 2017. LNCS, vol. 10319, pp. 219–236. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59746-1_12

11. El-Saber, N., Boronat, A.: BPMN formalization and verification using Maude. In: Proceedings of BM-FA, pp. 1–8. ACM (2014)
12. Goguen, J.A., Meseguer, J.: Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci. **105**(2), 217–273 (1992)
13. Güdemann, M., Poizat, P., Salaün, G., Ye, L.: VerChor: a framework for the design and verification of choreographies. IEEE Trans. Serv. Comput. **9**(4), 647–660 (2016)
14. Herbert, L., Sharp, R.: Using stochastic model checking to provision complex business services. In: Proceedings of HASE, pp. 98–105. IEEE (2012)
15. ISO/IEC: International Standard 19510, Information Technology - Business Process Model and Notation (2013)
16. Kheldoun, A., Barkaoui, K., Ioualalen, M.: Specification and verification of complex business processes - a high-level petri net-based approach. In: Motahari-Nezhad, H.R., Recker, J., Weidlich, M. (eds.) BPM 2015. LNCS, vol. 9253, pp. 55–71. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23063-4_4
17. Kossak, F.: A Rigorous Semantics for BPMN 2.0 Process Diagrams. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09931-6
18. Martens, A.: Analyzing web service based business processes. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 19–33. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31984-9_3
19. Mateescu, R., Salaün, G., Ye, L.: Quantifying the parallelism in BPMN processes using model checking. In: Proceedings of CBSE, pp. 159–168. ACM (2014)
20. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Compu. Sci. **96**(1), 73–155 (1992)
21. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Presicce, F.P. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-64299-4_26
22. Nguyen, H.N., Poizat, P., Zaïdi, F.: A symbolic framework for the conformance checking of value-passing choreographies. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q. (eds.) ICSOC 2012. LNCS, vol. 7636, pp. 525–532. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34321-6_36
23. Object Management Group: Business Process Model and Notation (BPMN) - V. 2.0, January 2011
24. Object Management Group: Decision Model and Notation Specification (DMN) - V. 1.1, May 2016
25. Poizat, P., Salaün, G.: Checking the realizability of BPMN 2.0 choreographies. In: Proceedings of SAC, pp. 1927–1934. ACM (2012)
26. Prandi, D., Quaglia, P., Zannone, N.: Formal analysis of BPMN via a translation into COWS. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 249–263. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68265-3_16
27. Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. J. Appl. Logic **10**(1), 2–31 (2012)
28. Raedts, I., Petkovic, M., Usenko, Y.S., van der Werf, J.M., Groote, J.F., Somers, L.: Transformation of BPMN models for behaviour analysis. In: Proceedings of MSVVEIS, pp. 126–137 (2007)
29. Rocha, C., Meseguer, J., Muñoz, C.: Rewriting modulo SMT and open system analysis. J. Log. Algebr. Methods Program. **86**(1), 269–297 (2017)
30. Viry, P.: Equational rules for rewriting logic. Theor. Comput. Sci. **285**(2), 487–517 (2002)

31. White, D.J.: Markov Decision Processes. Wiley, Chichester (1993)
32. Wong, P.Y.H., Gibbons, J.: A process semantics for BPMN. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 355–374. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88194-0_22
33. Wong, P., Gibbons, J.: Verifying business process compatibility. In: Proceedings of QSIC, pp. 126–131. IEEE (2008)
34. Wynn, M.T., Verbeek, H.M.W., van der Aalst, W.M.P., ter Hofstede, A.H.M., Edmond, D.: Business process verification - finally a reality! Bus. Process Manag. J. **15**(1), 74–92 (2009)

Associative Unification and Symbolic Reasoning Modulo Associativity in Maude

Francisco Durán[1], Steven Eker[2], Santiago Escobar[3](✉), Narciso Martí-Oliet[4], José Meseguer[5], and Carolyn Talcott[2]

[1] Universidad de Málaga, Málaga, Spain
duran@lcc.uma.es
[2] SRI International, Menlo Park, CA, USA
eker@csl.sri.com, clt@cs.stanford.edu
[3] Universitat Politècnica de València, València, Spain
sescobar@dsic.upv.es
[4] Universidad Complutense de Madrid, Madrid, Spain
narciso@ucm.es
[5] University of Illinois at Urbana-Champaign, Champaign, IL, USA
meseguer@illinois.edu

Abstract. We have added support for associative unification to Maude 2.7.1. Associative unification is infinitary, i.e., there are unification problems $u =^? v$ such that there is an infinite minimal set of unifiers, whereas associative-commutative unification is finitary. A unique feature of the associative unification algorithm implemented in Maude is that it is guaranteed to terminate with a *finite* and *complete* set of associative unifiers for a fairly large class of unification problems occurring in practice. For any problems outside this class, the algorithm returns a finite set of unifiers together with a *warning* that such set may be incomplete. This paper describes this associative unification algorithm implemented in Maude and also how other symbolic reasoning Maude features such as (i) variant generation; (ii) variant unification; and (iii) narrowing based symbolic reachability analysis have been extended to deal with associativity.

1 Introduction

Maude[1] is a language and a system based on rewriting logic [10]. Maude is a mathematical modeling language thanks to its logical basis and its initial model semantics, allowing it and its formal tool environment to be used in three, mutually reinforcing ways: as a declarative programming language, as an executable formal specification language, and as a formal verification system.

Automated reasoning features were addressed by some Maude predecessors, but never included in Maude until very recently: (i) Eqlog [22] envisioned an

[1] Maude is publicly available at http://maude.cs.illinois.edu.

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 98–114, 2018.
https://doi.org/10.1007/978-3-319-99840-4_6

integration of order-sorted equational logic with Horn logic, providing logical variables, constraint solving, and automated reasoning capabilities on top of order-sorted equational logic; and (ii) MaudeLog [30] envisioned an integration of order-sorted rewriting logic with queries including logical variables. Among the many automated reasoning features that could be integrated into Maude, we have focused on order-sorted equational unification and order-sorted narrowing-based reachability.

Order-sorted unification and narrowing modulo axioms became first available in 2009 as part of the Maude 2.4 release [9]. Unification became available as a built-in feature in Maude while narrowing was available in Full Maude, an extension of Maude written in Maude itself. Unification worked for any combination of symbols being either free or associative-commutative (AC). Narrowing worked for modules having only rules and axioms and relied on the built-in unification algorithm. It supported the concept of symbolic reachability analysis of terms with logical variables, computing suitable substitutions for the variables in both the origin and the destination terms [19].

Unification and narrowing were updated in 2011 as part of the Maude 2.6 release [14]. First, built-in unification was extended to allow any combination of symbols being either free, commutative (C), associative-commutative (AC), or associative-commutative with an identity symbol (ACU). The performance was dramatically improved, allowing further development of other techniques in Maude. Second, the concept of *variant* [11] was added to Maude. The introduction of variants led to a significant improvement in Maude's reasoning capabilities: variant generation, variant-based unification, and symbolic reachability based on variant-based unification became all available for the first time. However, all the variant-based features and the narrowing-based reachability were only available in Full Maude, and for a restricted class of theories called *strongly right irreducible*.

Unification and narrowing were extended again in 2016 as part of the Maude 2.7 release [13]. First, the built-in unification algorithm allows any combination of symbols being free, C, AC, ACU, CU (commutativity and identity), U (identity), Ul (left identity), and Ur (right identity). Second, variant generation and variant-based unification are implemented as built-in features in Maude. This built-in implementation works for any convergent theory modulo the axioms described above, both allowing very general equational theories (beyond the strongly right irreducible ones) and boosting the performance not only of these features but of their applications. Third, narrowing-based reachability is still only available in Full Maude, but uses the built-in variant-based unification.

In this paper we present our last addition to the built-in unification algorithms: *associative unification*. Furthermore, we explain how the range of the following additional symbolic reasoning features has thus been substantially extended with reasoning modulo associativity: (i) variant generation; (ii) variant unification; and (iii) narrowing based symbolic reachability analysis. All these newly extended features are now supported by Maude 2.7.1 [8].

At first sight, adding symbolic reasoning modulo associativity might seem to be an incremental contribution to Maude, but this is not at all the case. As we explain in this paper, not only are important new applications made now possible thanks to the above extended features (i)–(iii) but, more basically, since associative unification is infinitary in general, the development of an efficient and *effective in practice* associative unification algorithm that furthermore supports order-sorted typing and combination with any other symbols either free or themselves combining some A and/or C and/or U axioms, has been a highly non-trivial challenge. A key concern in meeting this challenge has been the identification of a fairly broad class of unification problems appearing in many practical applications for which our algorithm is guaranteed to terminate with a *finite* and *complete* set of unifiers. To deal with the unavoidable possibility that the given unification problem may have an infinite set of unifiers, when the problem is outside the class supported by the algorithm in a complete way, the algorithm returns a finite set of unifiers with an *explicit warning* that such a set may be incomplete. In a good number of applications where we have used these new associative symbolic features of Maude—for example, in the symbolic analysis of cryptographic protocols where some symbols are associative— unification problems falling outside the class supported by our algorithm in a complete way often do not even arise in practice. In other applications, for example, in very large classes of narrowing problems and of critical pair computations, we can determine *a priori* that they will fall within the complete class of problems supported by our algorithm.

In Sect. 2, we explain how the associative unification algorithm works. In Sect. 3, we show the built-in order-sorted unification extended to the associative case. In Sect. 4, we show how variant generation works with the associative case. In Sect. 5, we show how variant-based unification works with the associative case. In Sect. 6, we show how narrowing-based reachability works with the associative case. In Sect. 7, we provide additional experimental evidence for the practical effectiveness of the approach.

2 The Associative Unification Algorithm in Maude

The problem of elementary associative unification with free constants, otherwise known as solving equations in free semigroups, has a long history. The problem of deciding unifiability was finally solved in 1977 by Makanin [28]. This method has subsequently been simplified [3,40], extended to enumerate a complete sets of unifiers [26], and compute a finite encoding of a complete set of unifiers [24]. Efforts to improve the efficiency of Makanin's algorithm have reduced the complexity bound to EXPSPACE [23], and an alternative technique by Plandowski that works with compressed representations of words has resulted in a decision procedure that runs in PSPACE [33] and a solver that runs in DEXPTIME [34].

Despite this theoretical progress, and some stand alone implementations of Makanin's algorithm [1,2], systems that support unification modulo equational theories have eschewed associativity for several reasons: In order to be useful

in such a system, a unification algorithm must smoothly combine with unification algorithms for other theories. While simultaneous unification (needed for modern combination algorithms) can be reduced to single equation unification in the associative case [25,27], performing combination with finite encodings of infinite sets of unifiers rather than unifiers appears to be an open problem. Furthermore, typical applications for unification such as completion and narrowing normally only work with finite sets of unifiers. Finally Makanin's algorithm and its derivatives have high complexity, while Plandowski's approach, giving the best complexity bounds, is impractical since it essentially requires an encoding of the answer to be guessed.

2.1 The pig-pug procedure

In developing Maude's associative unification algorithm, we take a pragmatic approach in that we will only be complete on some subset of those problems which have a finite complete set of unifiers, and we will warn the user whenever we cannot guarantee completeness. We base the algorithm on a procedure for elementary associative unification variously referred to as PIG-PUG [3] or Plotkin's algorithm [35] which is complete, but which in general does not terminate. Other authors have extended this approach to handle special cases of associativity [17], including those arising in modal logics [5,38].

The PIG-PUG procedure starts with an elementary associative unification problem, i.e. a single equation, $x =^? y$ where x and y are words over a set of variables and an associative symbol "_._". It constructs a search tree where each state is an equation $x_1 \ldots x_n =^? y_1 \ldots y_m$, with the original problem being the root node. A state where both sides of the equation are the empty word corresponds to a success, whereas a state where one side of the unification problem is the empty word and the other side is some nonempty word corresponds to failure.

A state $x_1 \ldots x_n =^? y_1 \ldots y_m$ where x_i and y_i are non-empty words gives rise to one or more new states. If x_1 and y_1 are equal, then cancelation yields a single new state $x_2 \ldots x_n =^? y_2 \ldots y_m$. Otherwise three possibilities have to be considered:

1. The substitution $\sigma = \{x_1 \mapsto y_1\}$ is applied to both sides of the unification problem, yielding $y_1.\sigma(x_2) \ldots \sigma(x_n) =^? y_1.\sigma(y_2) \ldots \sigma(y_m)$. The leftmost variables cancel yielding a new state $\sigma(x_2) \ldots \sigma(x_n) =^? \sigma(y_2) \ldots \sigma(y_m)$.
2. The substitution $\psi = \{x_1 \mapsto y_1.x_1'\}$ is applied to both sides of the unification problem, yielding $y_1.x_1'.\psi(x_2) \ldots \psi(x_n) =^? y_1.\psi(y_2) \ldots \psi(y_m)$. The leftmost variables cancel yielding a new state $x_1'.\psi(x_2) \ldots \psi(x_n) =^? \psi(y_2) \ldots \psi(y_m)$.
3. The substitution $\rho = \{y_1 \mapsto x_1.y_1'\}$ is applied to both sides of the unification problem, yielding $x_1.\rho(x_2) \ldots \rho(x_n) =^? x_1.y_1'.\rho(y_2) \ldots \rho(y_m)$. The leftmost variables cancel yielding a new state $\rho(x_2) \ldots \rho(x_n) =^? y_1'.\rho(y_2) \ldots \rho(y_m)$.

When a success state is found, a unifier can be computed by composing the substitutions used along the path to the success state. In general the search tree

is infinite because while step (1) always decreases the number of symbols in the problem, steps (2) and (3) can increase the lengths of the words in nonlinear problems.

It is well known that if no variable occurs more than twice in the starting state, steps (2) and (3) cannot increase the number of symbols in the problem since the substitution adds at most two symbols and the cancelation removes two symbols. Because the number of variables is not increased in the new state of steps (2) and (3), the number of possible equations that can appear in the tree is finite, and a terminating algorithm can be obtained by cycle detection [39]. However the result is then a search graph which may contain cycles encoding infinite sets of unifiers.

We also observe that if nonlinear variables only occur on one side (say the right-hand side) of the initial equation PIG-PUG terminates. This is because none of the three steps can increase the number of symbols on the left-hand side of the equation, and while step (3) can increase the number of symbols on the right-hand side of the equation, it consumes a left-hand side symbol (say x), and so can be executed at most $|x|$ times, after which no further increase in the number of right-hand side symbols can occur. Steps (1) and (2) both consume right-hand side symbols so the right-hand side of the equation must eventually become exhausted if the left-hand side has not been exhausted first.

2.2 Forcing Termination

For a practical implementation, termination is required. However, the PIG-PUG search can be safely pruned in a couple of ways, that can often guarantee termination and depend on both being in an order-sorted setting and combining different unification algorithms.

Firstly, while we are computing unsorted unifiers, in practice, we are only interested in those unsorted unifiers that have one or more corresponding order-sorted unifiers. It often happens that a variable has a sort that is too low to take a term with the associative symbol on top. We call such variables *element variables*. An element variable X cannot be subject to a replacement $X \mapsto Y.X'$ since any resulting unifier would result in X being assigned a term with the associative symbol on top and such an unsorted unifier would not correspond to any order-sorted unifier.

Secondly, when elementary associative unification is used to deal with associative symbols occurring in a general unification problem, some of the variables are actually introduced to abstract terms from alien theories. We call such variables *abstraction variables*. If a variable X is introduced to abstract a subterm from a collapse-free alien theory F, then X cannot be subject to a replacement $X \mapsto Y.X'$ since that would produce an immediate theory clash between the associative theory and F. Nor can such a variable be subject to an assignment $X \mapsto Y$ if Y is abstracting some subterm from a different collapse-free alien theory G, since this would produce an immediate theory clash between F and G.

We refer to both element variables and abstraction variables for subterms from collapse-free alien theories as *constrained variables*. For the purpose of

determining termination using either criteria, constrained variables can be ignored. If termination cannot be guaranteed using these two criteria, we force termination by only exploring the search tree up to a finite depth heuristically determined from the input problem. In this case, we set a flag to acknowledge possible incompleteness. Likewise, if due to cycle detection we end up with a search graph that encodes an infinite set of unifiers, we return only some of the infinite set of unifiers (the acyclic unifiers) and set the incompleteness flag. That is, the incompleteness flag is set if we are forced to either prune unexplored branches or avoid traversing a cyclic graph encoding an infinite set of unifiers. This incompleteness flag can be trivially percolated through unification combination and other algorithms that use unification to warn the user.

2.3 Combining Unification Algorithms

Modern unification combination algorithms require simultaneous unification for each theory to avoid termination issues. To solve simultaneous associative unification problems we solve one equation at a time and substitute the solution into the current partial solution and the remaining equations. We simplify equations by canceling from the right and left ends of each unificand where possible, making use of constrained variables. We heuristically attempt to avoid incompleteness by careful choice of the next equation to solve; ideally we choose one for which our PIG-PUG implementation is complete; otherwise we prefer equations that do not share variables between the left- and right-hand sides.

At the top level, in the unification combination algorithm, we try to avoid incompleteness by delaying the solution of the simultaneous unification subproblems for associative symbols as long as possible, in the hope that bindings to variables shared with other theories will constrain the search.

3 Built-In Order-Sorted Unification Modulo Axioms

Maude currently provides an order-sorted Ax-unification algorithm for all order-sorted theories (Σ, Ax) such that the order-sorted signature Σ is *preregular* modulo Ax (see [16, Footnote 2]) and the axioms Ax associated to function symbols can have any combination (even empty) of the following equational attributes: the `assoc` attribute (A), the `comm` attribute (C), the `assoc comm` attributes (AC), the `assoc comm id` attributes (ACU), the `comm id` attributes (CU), the `id` attribute (U), the `left id` attribute (Ul), and the `right id` attribute (Ur). The remaining cases: `assoc id`, `assoc left id`, and `assoc right id` are excluded, but can be easily supported by using variant unification (see Sect. 6). Maude 2.7.1 provides an Ax-unification command of the form

```
unify [n] in ⟨ModId⟩ : ⟨Term-1⟩ =? ⟨Term'-1⟩ /\ ... /\ ⟨Term-k⟩ =? ⟨Term'-k⟩ .
```

where $k \geq 1$, n is an optional argument providing a bound on the number of unifiers requested, and `ModId` is the module where the command takes place.

The unification infrastructure now supports the notion of incomplete unification algorithms (e.g. for associative unification).

Let us show some examples of unification with an associative attribute. Consider a very simple module where the symbol _._ is associative:

```
fmod UNIFICATION-A is protecting NAT .
    sort NList .
    subsort Nat < NList .
    op _._ : NList NList -> NList [assoc] .
endfm
```

Even if associative unification is infinitary (we include concrete examples below) there are many realistic unification problems that are still finitary. The following unification problem returns five unifiers:

```
Maude> unify in UNIFICATION-A : X:NList . Y:NList . Z:NList =? P:NList . Q:NList .

Solution 1
X:NList --> #1:NList . #2:NList
Y:NList --> #3:NList
Z:NList --> #4:NList
P:NList --> #1:NList
Q:NList --> #2:NList . #3:NList . #4:NList

Solution 2
X:NList --> #1:NList
Y:NList --> #2:NList . #3:NList
Z:NList --> #4:NList
P:NList --> #1:NList . #2:NList
Q:NList --> #3:NList . #4:NList

Solution 3
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #3:NList . #4:NList
P:NList --> #1:NList . #2:NList . #3:NList
Q:NList --> #4:NList

Solution 4
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #3:NList
P:NList --> #1:NList . #2:NList
Q:NList --> #3:NList

Solution 5
X:NList --> #1:NList
Y:NList --> #2:NList
Z:NList --> #3:NList
P:NList --> #1:NList
Q:NList --> #2:NList . #3:NList
```

One possible condition for finitary associative unification (as explained in Sect. 2) is having linear *list* variables, as in the example above. On the other hand, the unification problem may not be linear, but it may be easy to detect that there is no unifier, e.g. it is impossible to unify a list X concatenated with itself with another list Y concatenated also with itself but with a natural number, e.g. 1, in between.

```
Maude> unify in UNIFICATION-A : X:NList . X:NList =? Y:NList . 1 . Y:NList .
No unifier.
```

When nonlinear variables occur on both sides of an associative unification problem, Maude will ensure termination (as explained in Sect. 2). Several cases are possible:

1. One or more cycles are detected, but they do not give rise to unifiers.

```
Maude> unify in UNIFICATION-A : 0 . Q:NList =? Q:NList . 1 .
No unifier.
```

2. There is at least one cycle that produces an infinite family of most general unifiers. In this case a warning will be issued and only the acyclic solutions are returned.

```
Maude> unify in UNIFICATION-A : 0 . X:NList =? X:NList . 0 .
Warning: Unification modulo the theory of operator _._ has encountered
an instance for which it may not be complete.

Solution 1
X:NList --> 0
Warning: Some unifiers may have been missed due to incomplete unification algorithm(s).
```

   Note that the unification problem `0 . X` $=^?$ `X . 0` has an infinite family of most general unifiers $\{\mathtt{X} \mapsto \mathtt{0}^n\}$ for $\mathtt{0}^n$ being a list of n consecutive `0` elements.

3. There is at least one nonlinear variable with more than two occurrences and Maude will use a depth bound rather than cycle detection. If the search tree grows beyond the depth bound, the offending branches will be pruned, and a warning will be given.

```
Maude> unify in UNIFICATION-A :
                    X:NList . X:NList . X:NList =? Y:NList . Y:NList . Z:NList . Y:NList .
Warning: Unification modulo the theory of operator _._ has encountered
an instance for which it may not be complete.

Solution 1
X:NList --> #1:NList . #1:NList . #1:NList . #1:NList
Y:NList --> #1:NList . #1:NList . #1:NList
Z:NList --> #1:NList . #1:NList . #1:NList

Solution 2
X:NList --> #1:NList . #1:NList . #1:NList
Y:NList --> #1:NList . #1:NList
Z:NList --> #1:NList . #1:NList . #1:NList

Solution 3
X:NList --> #1:NList . #1:NList
Y:NList --> #1:NList
Z:NList --> #1:NList . #1:NList . #1:NList
Warning: Some unifiers may have been missed due to incomplete unification algorithm(s).
```

See [8] for details on the meta-level commands for unification, which are extended with a new constant `noUnifierIncomplete`, and additional warnings generated during associative unification.

4 Built-In Variant Generation

Given an equational theory $(\Sigma, E \cup Ax)$ where $\overrightarrow{E}$ is a set of convergent oriented equations modulo the axioms Ax, the (E, Ax)-*variants* [11,21] of a term t are the pairs (u, σ), where σ is a substitution and u is the (E, Ax)-canonical form of $t\sigma$. A preorder relation of generalization that holds between such pairs provides a notion of most general variants and also of completeness of a set of variants. An equational theory has the *finite variant property* (or it is called a *finite variant theory*) iff there is a finite and complete set of most general variants for each term.

The equational theories that are admissible for variant generation in Maude 2.7.1 are as follows (despite being a finite variant theory). Let `fmod` $(\Sigma, E \cup Ax)$ `endfm` be an order-sorted functional module where E is a set of equations specified with the `eq` keyword and the attribute `variant`, and Ax is a set satisfying the restrictions explained in Sect. 3 for order-sorted Ax-unification. Furthermore, the equations E must be unconditional, not using the `owise` attribute, and confluent, terminating, sort-decreasing, and coherent modulo Ax (we then call the equational theory *convergent*).

Any system module `mod` $(\Sigma, G \cup E \cup Ax, R)$ `endm` where G is an additional set of equations and R is a set of rules, is also considered admissible for variant generation if the equational part $(\Sigma, E \cup Ax)$ satisfies the conditions described above. Note that Maude requires that the equations E used for variant generation (and variant-based unification) should be clearly distinguished from the standard equations G in Maude by using the attribute `variant` (both E and G are used for term simplification).

Maude provides a variant generation command of the form:

```
get variants [ n ] in ⟨ModId⟩ : ⟨Term⟩ .
```

where n is an optional argument providing a bound on the number of variants requested, so that if the cardinality of the set of variants is greater than the specified bound, the variants beyond that bound are omitted; and `ModId` is the module where the command takes place.

The variant generation infrastructure now supports the notion of incomplete unification algorithms (see Sect. 3). The adaptation of the previous algorithm in Maude 2.7 to incomplete unification algorithms was not very difficult, since variants are computed using the *folding variant narrowing strategy* [21] and each narrowing step calls the order-sorted equational unification algorithm of Sect. 3. Thus, incompleteness may result in a folding variant narrowing tree with infinite width being trimmed and losing some of the possible variants; see [8] for further details.

Let us show some examples of variant generation with an associative operator.

```
fmod VARIANT-ASSOC is protecting NAT .
  sort NList .
  subsort Nat < NList .
  op _._ : NList NList -> NList [assoc] .
  var E : Nat . var L : NList .

  ops tail prefix : NList -> NList .
  eq tail(E . L) = L [variant] .
  eq prefix(L . E) = L [variant] .

  ops head last : NList -> Nat .
  eq head(E . L) = E [variant] .
  eq last(L . E) = E [variant] .

  op duplicate : NList -> Bool .
  eq duplicate(L . L) = true [variant] .
endfm
```

Some terms have a finite set of most general variants modulo associativity.

```
Maude> get variants in VARIANT-ASSOC : head(prefix(tail(L))) .

Variant #1
Nat: head(prefix(tail(#1:NList)))
L --> #1:NList

Variant #2
Nat: head(prefix(%2:NList))
L --> %1:Nat . %2:NList

Variant #3
Nat: head(#2:NList)
L --> #1:Nat . #2:NList . #3:Nat

Variant #4
Nat: %3:Nat
L --> %1:Nat . %3:Nat . %4:NList . %2:Nat
```

However, some terms may hit incomplete associative unification calls (see Sect. 3), and an incompleteness warning for associative unification will be printed.

```
Maude> get variants in VARIANT-ASSOC : duplicate(prefix(L) . tail(L)) .

Variant #1
[Bool]: duplicate(prefix(#1:NList) . tail(#1:NList))
L --> #1:NList

Variant #2
[Bool]: duplicate(%1:NList . tail(%1:NList . %2:Nat))
L --> %1:NList . %2:Nat

Variant #3
[Bool]: duplicate(prefix(%1:Nat . %2:NList) . %2:NList)
L --> %1:Nat . %2:NList

Variant #4
[Bool]: duplicate(#1:Nat . #2:NList . #2:NList . #3:Nat)
L --> #1:Nat . #2:NList . #3:Nat

Variant #5
[Bool]: duplicate(#1:Nat . #2:Nat)
L --> #1:Nat . #2:Nat

Warning: Unification modulo the theory of operator _._ has encountered
an instance for which it may not be complete.
Variant #6
Bool: true
L --> %1:Nat . %1:Nat . %1:Nat

Variant #7
Bool: true
L --> %1:Nat . %1:Nat

No more variants.
Warning: Some variants may have been missed due to incomplete unification algorithm(s).
```

Note that the term `duplicate(prefix(L) . tail(L))` has an infinite set of most general variants for the case of returning the variant term `true`, i.e., the family of substitutions $\{$`L:NList` $\mapsto$ `N:Nat`...`N:Nat`$\}$. This is due to the associative unification call (`N:Nat . L:NList`) $=^?$ (`L:NList . N:Nat`) invoked internally by variant generation.

See [8] for details on the meta-level commands for variant generation, which are extended with a new constant `noVariantIncomplete`.

5 Built-In Variant-Based Unification

The most natural application of variant generation is unification in an equational theory $(\Sigma, E \cup Ax)$ where the equations E can be oriented into convergent rules

$\overrightarrow{E}$ modulo Ax. Intuitively, when we extend such an equational theory $(\Sigma, E \cup Ax)$ with a new equation `eq(x,x) = true`, two terms t and t' unify with substitution α modulo the equational theory if and only if $(\texttt{true}, \alpha)$ is a variant of the term $\texttt{eq}(t, t')$.

Given a module `ModId` satisfying the requirements of Sect. 4 and being a finite variant theory, Maude provides a command for equational unification:

```
variant unify [ n ] in ⟨ModId⟩ :  ⟨Term-1⟩ =? ⟨Term'-1⟩ /\ ... /\ ⟨Term-k⟩ =? ⟨Term'-k⟩ .
```

where $k \geq 1$ and n is an optional argument providing a bound on the number of unifiers requested, so that if the cardinality of the set of unifiers is greater than the specified bound, the unifiers beyond that bound are omitted.

The variant unification infrastructure now supports the notion of incomplete Ax-unification algorithms (see Sect. 3). Since, given a unification problem $u = v$, variant-based equational unification generates all the variants of u and v using the variant generation of Sect. 4 and then calls the order-sorted Ax-unification of Sect. 3 between each one of the variants of u and each one of the variants of v, incompleteness may arise in two possible ways: during variant generation, as illustrated in Sect. 4 with the generation of a warning, and during order-sorted unifier generation, as illustrated in Sect. 3 with the generation of a warning. Note that the warning is exactly the same in both contexts. We may print different warnings in the future, depending on where the incompleteness arose. That is, incompleteness may result in two folding variant narrowing trees with infinite width being trimmed, losing some of the possible variants, and also some Ax-unification calls returning a finite but incomplete set of unifiers.

Similarly to the incomplete variant generation in Sect. 4, we can have variant unification calls that cannot provide a finitary set of most general unifiers, using the `VARIANT-ASSOC` theory of Sect. 4. The unification problem `head(L) =? last(L)` $\wedge$ `prefix(L) =? tail(L)` has the same solutions as the unification problem `N:Nat . L:NList =? L:NList . N:Nat` with the family of solutions $\{$`L:NList` $\mapsto$ `N:Nat...N:Nat`$\}$, raising the incompleteness unification warning.

```
Maude> variant unify in VARIANT-ASSOC : head(L) =? last(L) /\ prefix(L) =? tail(L)  .

Warning: Unification modulo the theory of operator _._ has encountered
an instance for which it may not be complete.

Unifier #1
L --> %1:Nat . %1:Nat . %1:Nat

Unifier #2
L --> %1:Nat . %1:Nat

No more unifiers.
Warning: Some unifiers may have been missed due to incomplete unification algorithm(s).
```

See [8] for details on the meta-level commands for variant unification, which are extended with a new constant `noUnifierIncomplete`.

6 Narrowing-Based Symbolic Reachability Analysis

The modern application of narrowing, when the rules R are understood as *transition rules*, is that of *symbolic reachability analysis* [32]. Specifically, we consider

transition systems specified by order-sorted rewrite theories of the form `mod` $(\Sigma, E \cup Ax, R)$ `endm` where: (i) $E \cup Ax$ satisfies the requirements of Sect. 4, and (ii) the transition rules R are $E \cup Ax$-coherent and *topmost* (so that rewriting is always done at the top of the term). Then, narrowing modulo $E \cup Ax$ is a *complete* deductive method [32] for symbolic reachability analysis, i.e., for solving existential queries of the form $\exists \overline{x} : t \rightarrow^* t'$ where $\overline{x}$ are all the variables appearing in t and t', in the sense that the formula holds for $(\Sigma, E \cup Ax, R)$ iff there is a narrowing sequence $t \leadsto^*_{R,E\cup Ax} u$ such that u and t' have an $(E \cup Ax)$-unifier. The $(R, E \cup Ax)$-*narrowing* relation is defined as $t \leadsto_{\sigma,p,R,E\cup Ax} t'$ iff there is a non-variable position p of t, a (possibly renamed) rule $l \rightarrow r$ in R, and a unifier $\sigma \in \mathit{Unif}_{E\cup Ax}(t|_p, l)$ such that $t' = (t[r]_p)\sigma$.

The symbolic reachability infrastructure now supports the notion of incomplete unification algorithms (see Sects. 3 and 5). Incompleteness may result in a narrowing-based reachability tree with infinite width being trimmed and losing some of the possible solutions; see [8] for further details.

This symbolic reachability is supported by Full Maude's `search` command:

```
(search [ n,m ] in ⟨ModId⟩ : ⟨Term-1⟩ ⟨SearchArrow⟩ ⟨Term-2⟩ .)
```

where: n and m are optional arguments providing, respectively, a bound on the number of solutions and the maximum depth of the search; *`ModId`* is the module where the search takes place; *`Term-1`* is the starting term, which cannot be a variable, but may contain variables; *`Term-2`* is the term specifying the pattern that has to be reached (with some variables possibly shared with the starting term); and *SearchArrow* is an arrow indicating the form of the narrowing proof, where `~>1` indicates a narrowing proof consisting of exactly one step; `~>+` indicates a proof of one or more steps; `~>*` indicates a proof of none, one, or more steps; and `~>!` indicates that the reached term cannot be further narrowed.

Let us illustrate the power of performing narrowing-based reachability analysis modulo variant equations and axioms, including associativity. Consider the specification of a generic grammar interpreter in Maude, based on [4]. We define a symbol `_@_` to represent the interpreter configurations, where the first underscore represents the current string (of terminal and non-terminal symbols), and the second underscore stands for the considered grammar. For simplification, we provide four non-terminal symbols `S`, `A`, `B`, and `C` for sort `NSymbol` and four terminal symbols 0, 1, 2, and the finalizing mark `eps` (the empty string) for sort `TSymbol`, but a parametric specification would have been more appropriate.

```
(mod GRAMMAR is
 sorts Symbol NSymbol TSymbol String Production Grammar Conf .
 subsorts TSymbol NSymbol < Symbol < String .
 subsort Production < Grammar .
 ops 0 1 2 eps : -> TSymbol .
 ops S A B C : -> NSymbol .
 op _@_ : String Grammar -> Conf .
 op _->_ : String String -> Production .
 op __ : String String -> String [assoc id: eps] .
 op mt : -> Grammar .
 op _;_ : Grammar Grammar -> Grammar [assoc comm id: mt] .
 vars  L1 L2 U V : String .
 var G : Grammar .
```

```
 var N : NSymbol .
 var T : TSymbol .
 rl ( L1 U L2 @ (U -> V) ; G) => ( L1 V L2 @ (U -> V) ; G) .
endm)
```

Note the important fact that the string concatenation symbol `__` is not only `assoc`, but has also `eps` as its *identity* element. This means that in each narrowing step with the interpreter's rule equational unification must be performed modulo AU and not just modulo A. This is not supported by the order-sorted unification of Sect. 3. Therefore, the identity property is transformed into the variant equations:

```
eq eps U = U [variant] .      eq U eps V = U V [variant] .      eq V eps = V [variant] .
```

The interpreter can be used in two ways thanks to narrowing: to generate words of the given grammar, but also to parse a given string (see [7] for further references on this topic). Generating the words of a given grammar is defined by rewriting the configuration (`S` @ Γ) into (*st* @ Γ) where

st is a string of terminal symbols using the rules of the grammar Γ. For example, we have the following search query associated to a context-free grammar defining the language 0^n1^n:

```
Maude> (search [4] in GRAMMAR : S @ (S -> eps) ; (S -> 0 S 1)
                            ~>! U @ (S -> eps) ; (S -> 0 S 1) .)

Solution 1       Solution 2       Solution 3         Solution 4
U --> eps        U --> 0 1        U --> 0 0 1 1      U --> 0 0 0 1 1 1
```

Parsing a string *st* according to a given grammar Γ is defined by narrowing the configuration (N @ Γ) into (*st* @ Γ) where N is a logical variable denoting a non-terminal symbol. For example, we have the following search query:

```
Maude> (search [1] in GRAMMAR : N       @ (S -> eps) ; (S -> 0 S 1)
                            ~>* 0 0 1 1 @ (S -> eps) ; (S -> 0 S 1) .)
Solution 1
N --> S
```

Moreover, we can use narrowing to answer a more complex question: *What is the missing production so that the string "0 0 1" is parsed into the non-terminal symbol S?*

```
Maude> (search [1] in GRAMMAR : S     @ (N -> T) ; (S -> eps) ; (S -> 0 S 1)
                            ~>* 0 0 1 @ (N -> T) ; (S -> eps) ; (S -> 0 S 1) .)
Solution 1
N --> S ;
T --> 0
```

And we can use any grammar, e.g. a Type-0 grammar defining the language $0^n1^n2^n$.

```
Maude> (search [1] in GRAMMAR : N @ (S -> eps) ; (S -> 0 S B C) ; (C B -> B C) ;
                                    (0 B -> 0 1) ; (1 B -> 1 1) ; (1 C -> 1 2) ; (2 C -> 2 2)
                 ~>* 0 0 1 1 2 2 @ (S -> eps) ; (S -> 0 S B C) ; (C B -> B C) ;
                                    (0 B -> 0 1) ; (1 B -> 1 1) ; (1 C -> 1 2) ; (2 C -> 2 2) .)

Solution 1
N --> S
```

Note that we must restrict the search in the previous narrowing-based search commands, because narrowing does not terminate for these reachability problems. However, it is extremely important that no warning is shown, ensuring that the symbolic analysis is complete (w.r.t. associativity properties), despite associative unification being infinite for some uncommon cases. The key idea here is that associative variables (`L1`, `L2`, and `U`) in the transition rule are linear (`L1` and `L2`) or under order-sorted restrictions (`U`).

7 Conclusions and Related Work

Unification and narrowing in Maude have opened up many applications. Variant-based unification relies on the built-in unification algorithms in order to perform unification modulo a convergent theory. Several formal reasoning tools either rely on unification capabilities, such as termination proofs [15] and proofs of local confluence and coherence [16], or rely on narrowing capabilities such as narrowing-based theorem proving [37] or testing [36]. Also, narrowing-based reachability analysis has evolved into *logical model checking* [6,19], where standard model checking cannot handle either infinite sets of initial states or infinite sets of reachable states, but performing model checking from initial states with logical variables can handle these broader possibilities symbolically. The area of cryptographic protocol analysis has also benefited: the Maude-NPA tool [18] is the most successful example of combining narrowing and unification features in Maude; indeed Maude-NPA has already been tested with protocols using associative operators without any incomplete warning (see [20]). The Tamarin tool [29] also uses a variant-generation algorithm, initially only for the Diffie-Hellman theory, but recently extended to finite variant theories in Maude [12]. Finally, several decision procedures for formula satisfiability modulo equational theories have been provided based on narrowing [41] or by variant generation in finite variant theories [31]. All these applications could be substantially extended to the associative case, opening up new possibilities for symbolic reasoning.

Acknowledgements. Francisco Durán has been partially supported by Spanish MINECO/FEDER project TIN2014-52034-R and Univ. Málaga, Campus de Excelencia Internacional Andalucía Tech. Steven Eker was partially supported by NRL grant N00173-16-C-2005. Santiago Escobar was partially supported by the EU (FEDER) and the Spanish MINECO under grant TIN2015-69175-C4-1-R, by the Spanish Generalitat Valenciana under grant PROMETEOII/2015/013, and by the US Air Force Office of Scientific Research under award number FA9550-17-1-0286. Narciso Martí-Oliet has been partially supported by MINECO Spanish project TRACES (TIN2015–67522–C3–3R) and by Comunidad de Madrid program N-GREENS Software (S2013/ICE-2731). Jose Meseguer was partially supported by NRL under contract number N00173-17-1-G002. Carolyn Talcott was partially supported by ONR grants N0001415-1-2202, N00173-17-1-G002 and NRL grant N00173-16-C-2005.

References

1. Abdulrab, H.: Implementation of Makanin's algorithm. In: Schulz, K.U. (ed.) IWWERT 1990. LNCS, vol. 572, pp. 61–84. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55124-7_3
2. Abdulrab, H.: LOP: toward a new implementation of Makanin's algorithm. In: Abdulrab, H., Pécuchet, J.-P. (eds.) IWWERT 1991. LNCS, vol. 677, pp. 133–149. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56730-5_35
3. Abdulrab, H., Pécuchet, J.-P.: Solving word equations. J. Symbolic Comput. **8**(5), 499–521 (1989)
4. Alpuente, M., Cuenca-Ortega, A., Escobar, S., Meseguer, J.: Partial evaluation of order-sorted equational programs modulo axioms. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 3–20. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_1
5. Auffray, Y., Enjalbert, P.: Modal theorem proving: an equational viewpoint. J. Logic Comput. **2**(3), 247–295 (1992)
6. Bae, K., Escobar, S., Meseguer, J.: Abstract logical model checking of infinite-state systems using narrowing. In: van Raamsdonk, F. (ed.) 24th International Conference on Rewriting Techniques and Applications, RTA 2013, 24–26 June 2013, Eindhoven, The Netherlands, vol. 21. LIPIcs, pp. 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
7. Caballero, R., López-Fraguas, F.J.: A functional-logic perspective of parsing. In: Middeldorp, A., Sato, T. (eds.) FLOPS 1999. LNCS, vol. 1722, pp. 85–99. Springer, Heidelberg (1999). https://doi.org/10.1007/10705424_6
8. Clavel, M., et al.: Maude Manual (Version 2.7.1) (2016). http://maude.cs.illinois.edu
9. Clavel, M., et al.: Unification and narrowing in Maude 2.4. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 380–390. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02348-4_27
10. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
11. Comon-Lundh, H., Delaune, S.: The finite variant property: how to get rid of some algebraic properties. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 294–307. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_22
12. Dreier, J., Duménil, C., Kremer, S., Sasse, R.: Beyond subterm-convergent equational theories in automated verification of stateful protocols. In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 117–140. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_6
13. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Talcott, C.: Built-in variant generation and unification, and their applications in Maude 2.7. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 183–192. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_13
14. Durán, F., Eker, S., Escobar, S., Meseguer, J., Talcott, C.L.: Variants, unification, narrowing, and symbolic reachability in Maude 2.6. In: Schmidt-Schauß, M. (ed.) Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, 30 May–1 June 2011, Novi Sad, Serbia, vol. 10. LIPIcs, pp. 31–40. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)

15. Durán, F., Lucas, S., Meseguer, J.: Termination modulo combinations of equational theories. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS (LNAI), vol. 5749, pp. 246–262. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04222-5_15
16. Durán, F., Meseguer, J.: On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. J. Logic Algebraic Program. **81**(7–8), 816–850 (2012)
17. Enjalbcrt, P., Clerin-Debart, F.Ç.: A case of termination for associative unification. In: Abdulrab, H., Pécuchet, J.-P. (eds.) IWWERT 1991. LNCS, vol. 677, pp. 79–89. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56730-5_32
18. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: cryptographic protocol analysis modulo equational properties. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007-2009. LNCS, vol. 5705, pp. 1–50. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03829-7_1
19. Escobar, S., Meseguer, J.: Symbolic model checking of infinite-state systems using narrowing. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 153–168. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73449-9_13
20. Escobar, S., Meseguer, J., Meadows, C.: Maude-NPA manual v3.1 (2017). http://maude.cs.illinois.edu/w/index.php?title=Maude_Tools:_Maude-NPA
21. Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. J. Logic Algebraic Program. **81**(7–8), 898–928 (2012)
22. Goguen, J., Meseguer, J.: EQLOG: equality, types and generic modules for logic programming. In: DeGroot, D., Lindstrom, G. (eds.) Logic Programming, Functions, Relations and Equations, pp. 295–363. Prentice-Hall (1986)
23. Gutiérrez, C.: Satisfiability of word equations with constants is in exponential space. In: Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1998), pp. 112–119. IEEE Computer Society Press (1998)
24. Gutiérrez, C.: Solving equations in strings: on Makanin's algorithm. In: Lucchesi, C.L., Moura, A.V. (eds.) LATIN 1998. LNCS, vol. 1380, pp. 358–373. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054336
25. Hmelevskii, J.I.: Equations in free semigroups. Number 107 in Proceedings of the Steklov Institute of Mathematics, Moscow (1971)
26. Jaffa, J.: Minimal and complete word unification. J. ACM **37**(1), 47–85 (1990)
27. Lothaire, M.: Algebraic Combinatorics on Words. Number 90 in Encyclopedia of Mathematics and its Applications. Cambridge University Press (2002)
28. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Matematicheskii USSR Sbornik **32**(2), 129–198 (1977)
29. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 696–701. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_48
30. Meseguer, J.: Multiparadigm logic programming. In: Kirchner, H., Levi, G. (eds.) ALP 1992. LNCS, vol. 632, pp. 158–200. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0013826
31. Meseguer, J.: Variant-based satisfiability in initial algebras. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2015. CCIS, vol. 596, pp. 3–34. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29510-7_1
32. Meseguer, J., Thati, P.: Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. Higher-Order Symbolic Comput. **20**(1–2), 123–160 (2007)

33. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. J. ACM **51**(3), 483–496 (2004)
34. Plandowski, W.: An efficient algorithm for solving word equations. In: Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing (STOC 2006), pp. 467–476. ACM, New York (2007)
35. Plotkin, G.D.: Building in equational theories. Mach. Intell. **7**, 73–90 (1972)
36. Riesco, A.: Using big-step and small-step semantics in maude to perform declarative debugging. In: Codish, M., Sumii, E. (eds.) FLOPS 2014. LNCS, vol. 8475, pp. 52–68. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07151-0_4
37. Rusu, V.: Combining theorem proving and narrowing for rewriting-logic specifications. In: Fraser, G., Gargantini, A. (eds.) TAP 2010. LNCS, vol. 6143, pp. 135–150. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13977-2_12
38. Schmidt, R.A.: *E*-unification for subsystems of *S4*. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 106–120. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0052364
39. Schulz, K.U.: Makanin's algorithm for word equations-two improvements and a generalization. In: Schulz, K.U. (ed.) IWWERT 1990. LNCS, vol. 572, pp. 85–150. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55124-7_4
40. Schulz, K.U.: Word unification and transformation of generalized equations. J. Autom. Reasoning **11**(2), 149–184 (1993)
41. Tushkanova, E., Giorgetti, A., Ringeissen, C., Kouchnarenko, O.: A rule-based system for automatic decidability and combinability. Sci. Comput. Program. **99**, 3–23 (2015)

Proving Structural Properties of Sequent Systems in Rewriting Logic

Carlos Olarte[1], Elaine Pimentel[1], and Camilo Rocha[2](✉)

[1] Universidade Federal do Rio Grande do Norte, Natal, Brazil
carlos.olarte@gmail.com, elaine.pimentel@gmail.com
[2] Pontificia Universidad Javeriana, Cali, Colombia
camilo.rocha@javerianacali.edu.co

Abstract. General and effective methods are required for providing good automation strategies to prove properties of sequent systems. Structural properties such as admissibility, invertibility, and permutability of rules are crucial in proof theory, and they can be used for proving other key properties such as cut-elimination. However, finding proofs for these properties requires inductive reasoning over the provability relation, which is often quite elaborated, exponentially exhaustive, and error prone. This paper aims at developing automatic techniques for proving structural properties of sequent systems. The proposed techniques are presented in the rewriting logic metalogical framework, and use rewrite- and narrowing-based reasoning. They have been fully mechanized in Maude and achieve a great degree of automation when used on several sequent systems, including intuitionistic and classical logics, linear logic, and normal modal logics.

1 Introduction

Contemporary proof theory started with Gentzen's natural deduction and sequent calculus in the 1930's [7], and it has had a continuous development with the proposal of several proof systems for many logics. Proof systems are important tools for formalizing, reasoning, and analyzing structural properties of proofs, as well as determining computational and metalogical consequences of logical systems. As a matter of fact, proposing *good* calculi is one of the main research topics in proof theory.

It is more or less consensus that a good proof system should support the notion of *analytic proof* [5], where every formula that appears in a proof must be a sub-formula of the formulas to be proved. This restriction can be exploited to prove important metalogical properties of sequent systems such as consistency. In sequent systems, analyticity is often guaranteed by the *cut-elimination* property: if B follows from A and C follows from B, then C follows from A. That is, intermediate lemmas (e.g., B) can be "cut" from the proof system. It turns out that the proof of cut-elimination for a given system is often quite elaborated, exponentially exhaustive, and error prone. Hence the need for general

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 115–135, 2018.
https://doi.org/10.1007/978-3-319-99840-4_7

and effective methods for providing good automation strategies. In the case of cut-elimination, some of such methods strongly depend on the ability of showing *permutability* of rules that may depend on additional properties such as *admissibility* and *invertibility* of rules, which – in turn – may require induction-based reasoning.

Rewriting logic [6,15] is a *metalogical framework* that can be used to represent other logics and to reason about their metalogical properties [14]. When compared to a logical framework, a metalogical framework is more powerful because it includes the ability to reason about a logic's entailment relation as opposed to just being sound to simulate it. Moreover, important computational aspects of the logical system under study need to be encoded in flexible ways, so that such a system can become data, and be subject to transformations and efficient execution in a computational engine. Thanks to its reflective capabilities and initial reachability semantics, important inductive aspects of rewriting logic theories can be encoded in its own metalanguage so that theories, proofs, and provability can be mechanically analyzed with the help of rewriting logic systems such as Maude [6].

This paper develops new techniques, using rewriting logic as a metalogical framework, for reasoning about properties of sequent systems. Relying on rewrite- and narrowing-based reasoning, these techniques are introduced as procedures for proving admissibility, invertibility, and permutability of inference rules. Such procedures have been fully implemented in Maude. The case study analyses included in this paper comprise the following sequent systems: propositional intuitionistic logic (G3ip), multi-conclusion propositional intuitionistic logic (mLJ), propositional classical logic (G3cp), propositional linear logic (LL), and normal modal logics (K and S4). Beyond advocating for the use of rewriting logic as a metalogical framework, the novel algorithms presented here are able to automatically discharge many proof obligations and ultimately obtain the expected results.

The approach can be summarized as follows. The inference rules of a sequent system $\mathcal{S}$ are specified as (backward) rewrite rules modulo structural axioms (e.g., associativity, commutativity, and identity) in $\mathcal{R}_{\mathcal{S}}$, inducing a rewrite relation $\rightarrow_{\mathcal{S}}$ on multisets of sequents. From the rewriting logic viewpoint, the main results presented here are metatheorems about inductive reachability properties of $\rightarrow_{\mathcal{S}}$. These metatheorems entail sufficient conditions for proving inductive properties that can be generated and checked with the help of term rewriting and narrowing. More precisely, given an inductive property ϕ about $\mathcal{S}$, several subgoals ϕ_i are generated by unification modulo axioms. The system $\mathcal{S}$ is extended to $\mathcal{S}'$ by adding inductive lemmas as axioms and, if each ϕ_i can be $\rightarrow_{\mathcal{S}'}$-rewritten to the empty multiset, then ϕ holds in the initial reachability model of $\mathcal{S}$. In such a process, the original rewrite theory $\mathcal{R}_{\mathcal{S}}$ is extended and transformed in several ways: a painless task to implement thanks to the off-the-shelf reflective capabilities of rewriting logic supported by Maude. Ultimately, the resulting metatheorems can be seen as tactics for automating reasoning of sequent systems in rewriting logic. This approach is *generic* in the sense that

only mild restrictions are imposed on the formulas of the sequent system $\mathcal{S}$ and *modular* since properties can be proved incrementally.

Outline. The rest of the paper is organized as follows. Section 2 introduces the structural properties of sequent systems that are considered in this work and Sect. 3 presents order-sorted rewriting logic and its main features as a logical framework. Then, Sect. 4 establishes how to prove the structural properties based on a rewriting approach and Sect. 5 shows how to automate the process of proving these properties. Section 6 presents different sequent systems and properties that can be proved with the approach. Finally, Sect. 7 concludes the paper and presents some future research directions.

2 Three Structural Properties of Sequent-Based Logics

This section presents and illustrates three structural properties of sequent systems, namely, permutability, admissibility, and invertibility of rules. Notation and standard definitions are presented, which are illustrated with detailed examples.

Definition 1 (Sequent). *Let $\mathcal{L}$ be a formal language consisting of well-formed formulas. A* sequent *is an expression of the form $\Gamma \vdash \Delta$ where Γ (the* antecedent*) and Δ (the* succedent*) are finite multisets of formulas in $\mathcal{L}$, and $\vdash$ is the meta-level symbol of consequence. If the succedent of a sequent contains at most one formula, it is called* single-conclusion*, and* multiple-conclusion*, otherwise.*

Definition 2 (Sequent System). *A* sequent system $\mathcal{S}$ *is a set of rules of the form*

$$\frac{S_1 \quad \cdots \quad S_n}{S}\ r$$

where the sequent S is the conclusion *inferred from the* premise *sequents $S_1, \ldots, S_n$ in the rule r. If the set of premises is empty, then r is an* axiom. *In a rule introducing a connective, the formula with that connective in the conclusion sequent is the* principal formula*, and its sub-formulas in the premises are the* auxiliary formulas*. Systems with empty antecedents are called* one-sided*; otherwise they are called* two-sided.

As an example, Fig. 1 presents the two-sided single-conclusion propositional intuitionistic sequent system G3ip [21], with formulas built from the grammar:

$$F, G ::= p \mid \top \mid \bot \mid F \vee G \mid F \wedge G \mid F \supset G$$

where p is an atomic proposition. In this system, for instance, the conclusion $F \vee G$ of $\vee_L$ is the principal formula, while the formulas F and G are auxiliary formulas.

$$\frac{}{\Gamma, p \vdash p}\ I \qquad \frac{}{\Gamma \vdash \top}\ \top_R \quad \frac{\Gamma \vdash C}{\Gamma, \top \vdash C}\ \top_L \quad \frac{}{\Gamma, \bot \vdash C}\ \bot_L$$

$$\frac{\Gamma, F \vdash C \quad \Gamma, G \vdash C}{\Gamma, F \vee G \vdash C}\ \vee_L \qquad \frac{\Gamma \vdash F_i}{\Gamma \vdash F_1 \vee F_2}\ \vee_{R_i} \qquad \frac{\Gamma, F, G \vdash C}{\Gamma, F \wedge G \vdash C}\ \wedge_L \qquad \frac{\Gamma \vdash F \quad \Gamma \vdash G}{\Gamma \vdash F \wedge G}\ \wedge_R$$

$$\frac{\Gamma, F \supset G \vdash F \quad \Gamma, G \vdash C}{\Gamma, F \supset G \vdash C}\ \supset_L \quad \frac{\Gamma, F \vdash G}{\Gamma \vdash F \supset G}\ \supset_R$$

Fig. 1. System G3ip for propositional intuitionistic logic. In the I rule, p is atomic.

Definition 3 (Derivation). *A* derivation *in a sequent system $\mathcal{S}$ (called $\mathcal{S}$-derivation) is a finite labeled tree with nodes labeled by sequents and a single root, axioms at the top nodes, and where each node is connected with the (immediate) successor nodes (if any) according to the inference rules. A sequent S is* derivable *in the sequent system $\mathcal{S}$, denoted $\mathcal{S} \vdash S$, iff there is a derivation of S in $\mathcal{S}$. The system $\mathcal{S}$ is usually omitted when it can be inferred from the context.*

It is important to clearly distinguish the two different notions associated to the symbols $\vdash$ and $\vdash$ namely: the former is used to build sequents, while the latter (introduced in Definition 3) denotes derivability in a sequent system.

Definition 4 (Height of derivation). *The* height *of a derivation is the greatest number of successive applications of rules in it, where an axiom has height 0.*

The structural property of rule permutability [17,19] is stated next.

Definition 5 (Permutability). *Let r_1 and r_2 be inference rules in a sequent system $\mathcal{S}$. The rule r_2* permutes down *r_1, notation $r_2 \downarrow r_1$, if for every $\mathcal{S}$-derivation of a sequent S in which r_1 operates on S and r_2 operates on one or more of r_1's premises (but not on auxiliary formulas of r_1), there exists another $\mathcal{S}$-derivation of S in which r_2 operates on S and r_1 operates on zero or more of r_2's premises (but not on auxiliary formulas of r_2).*

For instance, consider the left $\vee_L$ and right $\vee_{R_i}$ rules for disjunction in G3ip. First, it can be observed that $\vee_L \downarrow \vee_{R_i}$ by using the following transformation:

$$\frac{\dfrac{\Gamma, F \vdash C_i \quad \Gamma, G \vdash C_i}{\Gamma, F \vee G \vdash C_i}\ \vee_L}{\Gamma, F \vee G \vdash C_1 \vee C_2}\ \vee_{R_i} \quad \rightsquigarrow \quad \frac{\dfrac{\Gamma, F \vdash C_i}{\Gamma, F \vdash C_1 \vee C_2}\ \vee_{R_i} \quad \dfrac{\Gamma, G \vdash C_i}{\Gamma, G \vdash C_1 \vee C_2}\ \vee_{R_i}}{\Gamma, F \vee G \vdash C_1 \vee C_2}\ \vee_L$$

The inverse permutation, however, does not hold, i.e., $\vee_{R_i} \not\downarrow \vee_L$. In fact, in the following derivation,

$$\frac{\dfrac{\Gamma, F \vdash C_i}{\Gamma, F \vdash C_1 \vee C_2}\ \vee_{R_i} \quad \Gamma, G \vdash C_1 \vee C_2}{\Gamma, F \vee G \vdash C_1 \vee C_2}\ \vee_L$$

derivability of $\Gamma, G \vdash C_1 \vee C_2$ does not imply derivability of $\Gamma, G \vdash C_i$; hence, such a derivation cannot start by applying the rule $\vee_{R_i}$.

Other two important structural properties are admissibility and invertibility.

Definition 6 (Admissibility and Invertibility). *Let $\mathcal{S}$ be a sequent system. An inference rule*

$$\frac{S_1 \quad \cdots \quad S_n}{S}$$

is called:

i. admissible in $\mathcal{S}$ *if S is derivable in $\mathcal{S}$ whenever $S_1, \ldots, S_n$ are derivable in $\mathcal{S}$.*
ii. invertible *in $\mathcal{S}$ if the rules $\frac{S}{S_1}, \ldots, \frac{S}{S_n}$ are admissible in $\mathcal{S}$.*

Proving invertibility often requires induction on the height of derivations, where all the possible rule applications have to be considered. For example, for proving that $\vee_L$ is invertible in G3ip, the goal is to show that both $\Gamma, F \vdash C$ and $\Gamma, G \vdash C$ are derivable whenever $\Gamma, F \vee G \vdash C$ is derivable. The result follows by a case analysis on the shape of the derivation of $\Gamma, F \vee G \vdash C$. Consider, e.g., the case when $C = A \supset B$ and the last rule applied is $\supset_R$, i.e., consider the following derivation:

$$\frac{\Gamma, F \vee G, A \vdash B}{\Gamma, F \vee G \vdash A \supset B} \supset_R$$

Then, by the inductive hypothesis, $\Gamma, F, A \vdash B$ and $\Gamma, G, A \vdash B$ are derivable and, by using $\supset_R$, the following holds:

$$\frac{\Gamma, F, A \vdash B}{\Gamma, F \vdash A \supset B} \supset_R \quad \text{and} \quad \frac{\Gamma, G, A \vdash B}{\Gamma, G \vdash A \supset B} \supset_R$$

as needed. On the other hand, $\vee_{R_i}$ is *not* invertible: if p_1, p_2 are different atomic propositions, then $p_i \vdash p_1 \vee p_2$ is derivable for $i = 1, 2$, but $p_i \vdash p_j$ is not for $i \neq j$.

In general, proving invertibility may involve some subtle details, as it will be seen in Sect. 6. A common one is the need for admissibility of the weakening structural rule. A *structural rule* does not introduce logical connectives, but instead changes the structure of the sequent. Since sequents are built from multisets, such changes are related to the cardinality of a formula or its presence/absence in a context. For example, the structural rules for *weakening* and *contraction* in the intuitionistic setting are:

$$\frac{\Gamma \vdash C}{\Gamma, \Delta \vdash C} \mathsf{W} \qquad \frac{\Gamma, \Delta, \Delta \vdash C}{\Gamma, \Delta \vdash C} \mathsf{C}$$

These rules are admissible in G3ip. The proof of admissibility of weakening is independent of any other results and it is also by induction on the height of derivations (and considering all possible rule applications).

Admissibility of contraction is more involved and often it depends on invertibility results. As an example, suppose that

$$\frac{\Gamma, F \vee G, F \vdash C \quad \Gamma, F \vee G, G \vdash C}{\Gamma, F \vee G, F \vee G \vdash C} \vee_L$$

Observe that the inductive hypothesis cannot be applied since the premises do not have duplicated copies of auxiliary formulas. In order to obtain a proof, invertibility of $\vee_L$ is needed: the derivability of $\Gamma, F \vee G, F \vdash C$ and $\Gamma, F \vee G, G \vdash C$ implies the derivability of $\Gamma, F, F \vdash C$ and $\Gamma, G, G \vdash C$; moreover, by the inductive hypothesis, $\Gamma, F \vdash C$ and $\Gamma, G \vdash C$ are derivable, and the result follows.

3 Rewriting Logic Preliminaries

This section briefly explains order-sorted rewriting logic [15] and its main features as a logical framework. Maude [6] is a language and tool supporting the formal specification and analysis of rewrite theories.

An *order-sorted signature* Σ is a tuple $\Sigma = (S, \leq, F)$ with a finite poset of sorts $(S, \leq)$ and a set of function symbols F typed with sorts in S, which can be subsort-overloaded. For $X = \{X_s\}_{s \in S}$ an S-indexed family of disjoint variable sets with each X_s countably infinite, the *set of terms of sort s* and the *set of ground terms of sort s* are denoted, respectively, by $T_\Sigma(X)_s$ and $T_{\Sigma,s}$; similarly, $T_\Sigma(X)$ and T_Σ denote the set of terms and the set of ground terms. 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 and such that $\theta(x) \in T_\Sigma(X)_s$ if $x \in X_s$, for any $x \in X$ and $s \in S$. A substitution θ is called *ground* iff $\theta(x) \in T_\Sigma$ or $\theta(x) = x$ for any $x \in X$. The application of a substitution θ to a term t is denoted by $t\theta$.

A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E \uplus B, R)$ with: (i) $(\Sigma, E \uplus B)$ an order-sorted equational theory with signature Σ, E a set of (possibly conditional) equations over T_Σ, and B a set of structural axioms – disjoint from the set of equations E – over T_Σ for which there is a finitary matching algorithm (e.g., associativity, commutativity, and identity, or combinations of them); and (ii) R a finite set of (possibly with equational conditions) rewrite rules over T_Σ. A rewrite theory $\mathcal{R}$ induces a rewrite relation $\rightarrow_\mathcal{R}$ on $T_\Sigma(X)$ defined for every $t, u \in T_\Sigma(X)$ by $t \rightarrow_\mathcal{R} u$ if and only if there is a rule $(l \rightarrow r \textbf{ if } \phi) \in R$ and a substitution $\theta : X \longrightarrow T_\Sigma(X)$ satisfying $t =_{E \uplus B} l\theta$, $u =_{E \uplus B} r\theta$, and $\phi\theta$ is (equationally) provable from $E \uplus B$ [2].

Appropriate requirements are needed to make an equational theory $\mathcal{R}$ *executable* in Maude. It is assumed that the equations E can be oriented into a set of (possibly conditional) sort-decreasing, operationally terminating, and confluent rewrite rules $\overrightarrow{E}$ modulo B [6]. For a rewrite theory $\mathcal{R}$, the rewrite relation $\rightarrow_\mathcal{R}$ is undecidable in general, even if its underlying equational theory is executable, unless conditions such as coherence [22] are given (i.e., rewriting with $\rightarrow_{R/E \uplus B}$ can be decomposed into rewriting with $\rightarrow_{E/B}$ and $\rightarrow_{R/B}$). The executability of a rewrite theory $\mathcal{R}$ ultimately means that its mathematical and execution semantics coincide.

The rewriting logic specification of a sequent system $\mathcal{S}$ is a rewrite theory $\mathcal{R}_\mathcal{S} = (\Sigma_\mathcal{S}, E_\mathcal{S} \uplus B_\mathcal{S}, R_\mathcal{S})$ where: $\Sigma_\mathcal{S}$ is an order-sorted signature describing the syntax of the logic $\mathcal{S}$; $E_\mathcal{S}$ is a set of executable equations modulo $B_\mathcal{S}$ corresponding to those parts of the deduction process that, being deterministic, can be safely automated as *computation rules* without any proof search; and $R_\mathcal{S}$ is a set of executable rewrite rules modulo $B_\mathcal{S}$ capturing those non-deterministic aspects of logical inference in $\mathcal{S}$ that require proof search. The point is that although both the computation rules $E_\mathcal{S}$ and the deduction rules $R_\mathcal{S}$ are executed by rewriting modulo the set of structural axioms $B_\mathcal{S}$, by the executability assumptions on $\mathcal{R}_\mathcal{S}$, the rewrite relation $\rightarrow_{E_\mathcal{S}/B_\mathcal{S}}$ has a single outcome in the form of a canonical form and thus can be executed blindly with "don't care" non-determinism and without any proof search. Furthermore, $B_\mathcal{S}$ provides yet one more level of computational automation in the form of $B_\mathcal{S}$-matching and $B_\mathcal{S}$-unification algorithms. This interplay between axioms, equations, and rewrite rules can ultimately make the specification $\mathcal{R}_\mathcal{S}$ very efficient with modest memory requirements.

4 Checking Admissibility, Invertibility, and Permutability

This section presents rewrite- and narrowing-based techniques for proving admissibility, invertibility, and permutability in sequent systems. They are presented as metatheorems about sequent systems – with the help of rewrite-based scaffolding such as terms and substitutions – and provide sufficient conditions for proving the desired properties.

The techniques introduced in this section assume that a sequent system $\mathcal{S}$ is a set of inference rules with sequents in the set $T_{\Sigma_\mathcal{S}}(X)$, where $\Sigma_\mathcal{S}$ is an order-sorted signature (see Section 3). The expression $\mathcal{S}_1 \cup \mathcal{S}_2$ denotes the extension of the sequent system $\mathcal{S}_1$ by adding the inference rules of $\mathcal{S}_2$ (and *vice versa*); in this case, the sequents in the resulting sequent system $\mathcal{S}_1 \cup \mathcal{S}_2$ are terms in the signature $\Sigma_{\mathcal{S}_1} \cup \Sigma_{\mathcal{S}_2}$. By an abuse of notation, for $\mathcal{S}$ a sequent system and S a sequent, the expression $\mathcal{S} \cup \{S\}$ denotes the sequent system obtained from $\mathcal{S}$ by adding the sequent S as an axiom, understood as a zero-premise rule. This convention is extensively used in the main results of this section. Finally, given a term $t \in T_{\Sigma_\mathcal{S}}(X)$, with $\Sigma_\mathcal{S} = (S, \leq, F)$, $\overline{t} \in T_{(S,\leq,F\cup C_{\overline{t}})}(X)$ is the term obtained from t by turning each variable $x \in vars(t)$ of sort $s \in S$ into the fresh constant $\overline{x}$ of sort s and where $C_{\overline{t}} = \{\overline{x} \mid x \in vars(t)\}$

It is assumed the existence of a unification algorithm for multisets (or sets) of sequents. Given two sequent terms S and T built from a signature $\Sigma_\mathcal{S}$ and structural axioms $B_\mathcal{S}$, the expression $CSU_{B_\mathcal{S}}(S,T)$ denotes the *complete set of unifiers* of S and T modulo $B_\mathcal{S}$. Recall that $CSU_{B_\mathcal{S}}(S,T)$ satisfies that, for each substitution $\sigma : X \longrightarrow T_\Sigma(X)$, there are substitutions $\theta \in CSU_{B_\mathcal{S}}(S,T)$ and $\gamma : X \longrightarrow T_\Sigma(X)$ such that $\sigma =_{B_\mathcal{S}} \theta\gamma$. Note that for a combination of free and associative and/or commutative and/or identity axioms $B_\mathcal{S}$, except for symbols that are associative but not commutative, such a finitary unification algorithm exists. In the development of this section, the expression CSU is used as an abbreviation for $CSU_{B_\mathcal{S}}$, where $B_\mathcal{S}$ are the structural axioms for sets/multisets of sequents.

Definition 7 introduces a notion of admissibility of a rule relative to another rule.

Definition 7 (Local admissibility). *Let* $\frac{S_1}{S}\ r_s$ *be a rule,* $\mathcal{S}$ *be a sequent system and* $\frac{T_1\ \cdots\ T_n}{T}\ r_t$ *be an inference rule in* $\mathcal{S}$*. The rule* r_s is admissible relative to r_t in $\mathcal{S}$ *iff for each* $\theta \in CSU(S_1, T)$*:*

$$\mathcal{S} \cup \left\{\overline{T_j\theta} \mid j \in 1..n\right\} \cup \bigcup_{j \in 1..n} \{\overline{S\gamma} \mid \gamma \in CSU(S_1, \overline{T_j\theta})\} \vdash \overline{S\theta},$$

where the variables in S *and* T *are assumed disjoint.*

For proving admissibility of the rule r_s, the goal is to prove that if S_1 is derivable, then S is derivable. The proof follows by induction on the height of a derivation π of S_1 (see Sect. 2). Suppose that the last rule applied in π is r_t. This is only possible if S_1 and T "are the same", up to substitutions. Hence, the idea is that each unifier θ of S_1 and T covers the cases where the rule r_t can be applied on the sequent S_1; different proof obligations are generated for each unifier. Consider, for instance, the proof obligation of the ground sequent $\overline{S\theta}$ for a given $\theta \in CSU(S_1, T)$. Namely, assume as hypothesis that the derivation below is valid in order to show that the sequent $\overline{S\theta}$ is provable:

$$\frac{T_1\theta \quad \cdots \quad T_n\theta}{S_1\theta}\ r_t \tag{1}$$

This means that all the premises in (1) should be assumed derivable. This is the purpose of extending the sequent system with the set of ground sequents $\left\{\overline{T_j\theta} \mid j \in 1..n\right\}$, interpreted here as axioms, in Definition 7. Moreover, by induction, it can be assumed that the theorem (i.e., S_1 implies S) is valid for the premises of (1) (note that such premises have a shorter derivation compared to the derivation of $S_1\theta$). Therefore, the following set of sequents can also be assumed as derivable and, thus, are added as axioms:

$$\bigcup_{j \in 1..n} \{\overline{S\gamma} \mid \gamma \in CSU(S_1, \overline{T_j\theta})\}$$

If, from the extended sequent system it is possible to show that the ground sequent $\overline{S\theta}$ is derivable, then the theorem will work for the particular case when r_t is the last applied rule in the derivation π of S_1. Since a complete set of unifiers is finite for sequents (as assumed in this section for any sequent system $\mathcal{S}$), then there are finitely many proof obligations to discharge in order to check if a rule is admissible relative to a rule in a sequent system. Observe that the set $CSU(S, T)$ may be empty. In this case, the set of proof obligations is empty and the property vacuously holds.

Theorem 1 presents sufficient conditions for the admissibility of a rule in a sequent system based on the notion of admissibility relative to a rule.

Theorem 1. *Let $\mathcal{S}$ be a sequent system and* $\dfrac{S_1}{S}\ r_s$ *be an inference rule. If r_s is admissible relative to each r_t in $\mathcal{S}$, then r_s is admissible in $\mathcal{S}$.*

Proof. Assume that S_1 is derivable in the system $\mathcal{S}$. The proof proceeds by induction on the height of such a derivation with case analysis on the last rule applied. Assume that the last applied rule is r_t. By hypothesis (using Definition 7), it can be concluded that S is derivable and the result follows.

The following definition introduces a notion of invertibility of a rule relative to another rule.

Definition 8 (Local invertibility). *Let $\mathcal{S}$ be a sequent system, and let* $\dfrac{S_1 \cdots S_m}{S}\ r_s$ *and* $\dfrac{T_1 \cdots T_n}{T}\ r_t$ *be inference rules in $\mathcal{S}$. The rule r_s* is invertible relative to r_t *iff for each $\theta \in CSU(S,T)$ and $1 \leq l \leq m$:*

$$\mathcal{S} \cup \left\{\overline{T_j\theta} \mid j \in 1..n\right\} \cup \bigcup_{i \in 1..m} \bigcup_{j \in 1..n} \{\overline{S_i\gamma} \mid \gamma \in CSU(S, \overline{T_j\theta})\} \vdash \overline{S_l\theta},$$

where the variables in S and T are assumed disjoint.

For checking invertibility of a rule r_s, the goal is to check that derivability is not lost when moving from the conclusion S to the premises S_l. The proof is by induction on the derivation π of S. Suppose that the last rule applied in π is r_t. For this to happen at the first place, S and T must unify. Then, for each $\theta \in CSU(S,T)$, the premise sequents $\overline{T_j\theta}$ of r_t are assumed to be derivable (and used to extend $\mathcal{S}$ with new axioms). Moreover, each ground term $\overline{S_i\gamma}$ can also be used as an inductive hypothesis since any application of r_s on $\overline{T_j\theta}$ has a shorter derivation than that of $\overline{T\theta}$. If, from all this in addition to the rules in $\mathcal{S}$, it is possible to prove derivable the premises S_l for all $1 \leq l \leq m$, then the theorem will work for the particular case where r_t was the last applied rule in the derivation π of S.

If the set $CSU(S,T)$ is empty, this means that the rules r_t and r_s cannot be applied on the same sequent and the property vacuously holds. For instance, consider the system G3ip in Fig. 1: the proof of invertibility of $\wedge_R$ does not need to consider the case of invertibility relative to $\vee_R$ since it is not possible to have, at the same time, a conjunction and a disjunction on the succedent of the sequent. In other logics as, e.g., G3cp (see Sect. 6.3), this proof obligation is certainly not vacuously discarded.

Theorem 2 presents sufficient conditions for checking the invertibility of a rule in a sequent system. The proof is similar to the one given for Theorem 1.

Theorem 2. *Let $\mathcal{S}$ be a sequent system and r_s an inference rule in $\mathcal{S}$. If r_s is invertible relative to each r_t in $\mathcal{S}$, then r_s is invertible in $\mathcal{S}$.*

This section is concluded by establishing conditions to prove permutability of rules.

Theorem 3. *Let $\mathcal{S}$ be a sequent system and* $\frac{S_1 \cdots S_m}{S}\, r_s$, $\frac{T_1 \cdots T_n}{T}\, r_t$ *be inference rules in $\mathcal{S}$. Then $r_s \downarrow r_t$ iff for each $\theta \in CSU(S,T)$, $1 \leq i \leq m$, $\gamma \in CSU(T, \overline{S_i\theta})$, and $1 \leq l \leq n$:*

$$\mathcal{S} \cup \left\{\overline{T_j\gamma} \mid j \in 1..n\right\} \cup \left\{\overline{S_k\theta} \mid k \in 1..m \wedge k \neq i\right\} \vdash \overline{T_l\theta},$$

where the variables in S and T are assumed disjoint.

Proof. Checking permutability does not require induction but a proof transformation. First of all, r_s, r_t should be applied to the conclusion sequent, hence all unifiers between the conclusions S and T are considered. Second, different cases need to be considered when r_t can be applied to one of the premises of r_s. Thus there is a proof obligation for each premise $S_i\theta$ where r_t can be applied. In each of such proof obligations the goal is to show that the premises of r_t are derivable ($T_l\theta$ on the right). For that, it can be assumed that the premises of r_t applied to the given premise of r_s are derivable ($T_j\gamma$ expression). Moreover, all the other premises of r_s are also assumed as derivable ($S_k\theta$ expression). If, from all these ground sequents and the rules in $\mathcal{S}$ it can be proved that T_l is derivable, for each $l = 1..n$, then $r_s \downarrow r_t$.

5 Reflective Implementation

The design and implementation of a prototype that offers support for the narrowing procedures introduced in Sect. 4 is discussed. The reader is referred to http://subsell.logic.at/theorem-maude for the implementation and the experiments summarized in Sect. 6.

5.1 Sequent System Specification

The reflective implementation relies on the following functional module that needs to be realized by the object-logic (i.e., the system to be analyzed):

```
fmod OBJ-LOGIC is
  --- Sequents and multisets of sequents
  sorts Sequent SSequent .
  subsort Sequent < SSequent .
  --- Building sequents
  op proved : -> Sequent [ctor] .
  op _,_ : SSequent SSequent -> SSequent [ctor assoc comm id: proved] .
endfm
```

The sort `Sequent` is used to represent sequent terms and the sort `SSequent` for representing multisets of sequent terms separated by comma. The constant `proved` is the identity of the multiset constructor and represents the empty sequent (i.e., no goals need to be discharged).

When formalizing a sequent system $\mathcal{S}$ as a rewrite theory $\mathcal{R}_\mathcal{S}$ there are two options (backwards or forwards) for expressing an inference rule as rewrite rule. In this paper, the backwards reasoning option is adopted, which rewrites the

target goal of an inference system to its premises. Hence, for instance, the rule $\wedge_L$ in G3ip will be expressed as a rewrite rule of the form $\Gamma, F \wedge G \vdash C \to \Gamma, F, G \vdash C$. The implementation assumes also a specific encoding for the inference rules as follows.

Definition 9. (Encoding logical rules). *A sequent rule* $\dfrac{S_1 \cdots S_m}{S}\ r_s$ *is encoded in the reflective implementation as:*
rl [rs] : S => proved . *if* $m = 0$*; and*
rl [rs] : S => S1, ..., Sm . *if* $m > 0$.

The first case in the encoding of logical rules corresponds to the case of an axiom, i.e., an inference rule without premises. The constant `proved` denotes the fact that an instance of an axiom is derivable by definition. The second case corresponds to those rules that have premises that need to be proved derivable.

The implementation requires a module with any (reasonable) concrete syntax for formulas and sequents, and adhering to the encoding of inference rules above. For instance, the following snippet of code specifies the syntax for the system G3ip:

```
fmod FORMULA-PROP is
  --- Atomic propositions, Formulas and sets of formulas
  sorts Prop Formula SFormula .
  subsort Prop < Formula < SFormula .
  op p : Nat -> Prop [ctor] . --- atomic Propositions
  ops False True : -> Formula [ctor] . --- False and True
  ops _-->_ _/\_ _\/_ : Formula Formula -> Formula [ctor] . --- connectives
  --- Building sets of formulas
  op * : -> SFormula . --- empty set of formulas
  op _;_ : SFormula SFormula -> SFormula [prec 40 ctor assoc comm id: * ] .
  eq F:Formula ; F:Formula = F:Formula . --- idempotency
endfm
```

The following module extends the module `OBJ-LOGIC` and specifies the inference rules of G3ip.

```
mod G3ip is
  pr FORMULA-PROP .
  inc OBJ-LOGIC .
  --- Constructor for sequents .
  op _|--_ : SFormula SFormula -> Sequent [ctor prec 50 format(b o r o )] .
  --- Rules
  rl [I] : P ; C  |-- P => proved .
  rl [AndL] : F /\ G ; C |-- H => F ; G ; C |-- H .
  rl [AndR] : C |-- F /\ G => (C |-- F) , (C |-- G) .
  rl [ImpL] : C ; F --> G |-- H => (C ; F --> G |-- F) , (C ; G |-- H) .
  ...
  op ANY : -> SFormula [ctor].
endm
```

The constant `ANY` is used to deal with extra-variables on the right-hand side of the rules, as it will be shown in an example below.

5.2 Property Specification

The reflective implementation uses the following theory to specify the input to the analysis task, i.e., the sequents to be proved derivable:

```
th TH-INPUT is
    pr META-LEVEL .
    --- Name of the module with the object-logic description
    op modName : -> Qid .
    --- List of theorems (hypotheses for the analyses)
    op knownTheorems : -> RuleSet .
    --- List of invertible rules
    op knownInvRules : -> QidList .
endth
```

Such a theory specifies the name of the module to be analyzed, the already proved theorems (e.g., admissibility of a given structural rule) and the rules that have been already proved to be invertible. As an example, the following snippet of code shows the implementation of the theory `TH-INPUT` for the module `G3ip`:

```
mod G3ip-TEST is
    ops modName seqType : -> Qid .  --- Name of the module to be analyzed
    eq modName = 'G3ip .
    op knownTheorems : -> RuleSet . --- Previously proved lemmas
    eq knownTheorems = none .
    op knownInvRules : -> QidList . --- Known invertible rules
    eq knownInvRules = nil .
    --- Theorems to be proved
    op Th-Weakening : -> Rule .     --- Admissibility of weakening
    eq Th-Weakening = ( rl '_|-_['C:SFormula,'F:Formula] => '
                          _|-_['_;_['C:SFormula,'ANY.SFormula],'F:Formula]
                            [ label('Th-Weakening) ]. ) .

  [...]
endm
```

As noted in Sect. 4, the properties of interest are specified by a sequent system $\mathcal{S}$ and an inference rule r. Given a rewrite theory $\mathcal{R}_{\mathcal{S}}$ representing $\mathcal{S}$, the inference rule r to be checked admissible, invertible, or permutable in $\mathcal{S}$ is represented by a rewrite rule, expressed as a meta-term, in the syntax of $\mathcal{S}$. For instance, the statement of the theorem for invertibility of $\wedge_R$ is generated with the aid of the auxiliary definition

```
op buildInvTheorem : Qid -> Rule .
```

that given the identifier of the rule (`'AndR'`, in this case) returns the following rule:

```
rl '_|-_['C:SFormula,'_/\_['F:Formula,'G:Formula]] =>
   '_`,_['_|-_['C:SFormula,'F:Formula],'_|-_['C:SFormula, 'G:Formula]]
   [label('Th-AndR)] .
```

`Th-AndR` is the meta-representation of the rule

```
rl [And] : C  |-- F /\ G => ( C |-- F , C |-- G) .
```

This is a very flexible way of encoding the theorems to be proved. For instance, in order to use the inductive hypothesis on a sequent T_j, it suffices to rewrite `Th-AndR` on T_j, thus resulting in the needed (derivable) sequents/axioms (see e.g., the term $\{\overline{S_i\gamma} \mid \gamma \in CSU(S, \overline{T_j\theta})\}$ in Definition 8).

Special care needs to be taken when the inference rule to check has extra variables in the premises. In general, the rewrite rule associated to such an inference rule would have extra variables in the right-hand side and could not be used for execution (unless a strategy is provided). Nevertheless, these extra variables can be encoded as fresh constants, yielding a rewrite rule that is executable. This is exemplified in the theorem for admissibility of Weakening in module `G3ip-TEST` that uses the constant `ANY` defined in the module `G3ip`. Note that `Th-Weakening` is just the meta-representation of

```
rl [Th-Weakening] : C |-- F =>  C ; ANY |-- F .
```

It is worth noticing that this rewriting rule is written from the premise to the conclusion (see rule W in Sect. 2). The reason is that the proof of admissibility requires to show that assuming the premise of the rule, the conclusion is valid (see Definition 7).

5.3 The Algorithms

The reflective implementation follows closely the definitions of the previous section. It offers functions that implement algorithms for each one of the theorems in Sect. 4; for sequent system $\mathcal{R}_{\mathcal{S}}$ and rule r:

`admissible?` checks if r is admissible in $\mathcal{S}$ by validating the conditions in Theorem 1.
`invertible?` checks if r is invertible in $\mathcal{S}$ by validating the conditions in Theorem 2.
`permutes?` checks if r permutes in $\mathcal{S}$ by validating the conditions in Theorem 3.

The output of each one of these algorithms is a list of tests, one per rule in $\mathcal{S}$. The test for a rule r_t indicates whether r has the desired property relative to r_t. Take for instance the procedure:

```
op invertible? : Qid -> Bool .
eq invertible?(Q) = resultTrue(analyze(buildInvTheorem(Q))) .
```

Given the identifier of a rule Q, it first builds the invertibility theorem, generates and executes all the needed proof obligations (`analyze(.)`) and returns `true` only if all the proof obligations succeed (`resultTrue`).

The procedure `analyze` tests the given rule Q against all the rules defined in the module. It uses the auxiliary function:

```
op holds? : Rule Qid -> Bool .
```

that computes the set of unifiers by using the Maude function `metaDisjointUnify` and checks the conditions described in Sect. 4. For that, operations on the `META-LEVEL` are used to, e.g., extend the module with the needed axioms (rewriting rules when $m = 0$ in Definition 9) and transform variables into constants. Moreover, the `metaSearch` procedure is used to check the entailment in, e.g., Definition 7.

Since the entailment relation is, in general, undecidable, all the tests are performed up to a given search depth and, when it is reached, the procedure returns `false`. Hence the procedures are sound (in the sense of the theorems in Sect. 4) but not complete (due to the undecidability of the logic and the fact that the goals are inductive properties).

Finally, the implementation also includes macros based on these algorithms, e.g., `analyzePermutation` for checking the permutation status of all rules.

6 Case Studies

This section presents properties of several sequent systems that can be automatically checked with the algorithms presented in Sect. 5. The general idea is that, given a sequent system $\mathcal{S}$ and a sequent S representing an admissibility, invertibility, or permutability problem instance, the experiments in this section use the encoding for $\mathcal{S}$ and S (Sect. 5) – and the rewriting logic framework – to check if S is derivable in $\mathcal{S}$, as follows:

$$\mathcal{S} \vdash S \qquad \text{if} \qquad enc(S) \rightarrow^{*}_{enc(\mathcal{S})} \texttt{proved},$$

where $enc(\mathcal{S})$ and $enc(S)$ denote, respectively, the encoding of $\mathcal{S}$ and S.

For each calculi, the results about invertibility and admissibility of the structural rules W (weakening) and C (contraction), and permutability are summarized in a table using the following conventions:

- $\surd_{\texttt{T}}$ means that the property holds for the given system and the tool is able to prove it (thus returning `true`).
- $\surd_{\texttt{F}}$ means that the property does not hold for the given system and the tool returns `false`.
- $\sim_{\texttt{DN}}$ means that the property holds but the tool was not able to prove it (then returning `false`).

6.1 System G3ip

An important remark is that propositional intuitionistic logic is decidable. However, since the rule $\supset_L$ replicates the principal formula in the left premise, a careless specification of this rule can result in infinite computations. For instance, the sequent $p \supset q \vdash q$ is not provable. However, a proof search trying to rewrite that sequent into `proved` will generate the infinite chain of goals $(p \supset q \vdash p), (p \supset q \vdash p), (p \supset q \vdash p), \cdots$.

One solution for this problem is to consider *sets* instead of multisets of sequents (i.e., by adding an equation for idempotency in the module `SEQUENT`). This solution is akin to the procedure of detecting whether a sequent in a derivation tree is equal to one of its predecessors. In this way a complete decision procedure for propositional intuitionistic logic can be obtained.

The results for invertibility of rules and admissibility of structural rules for $\mathsf{G3ip}$ are summarized below.

Invertibilities											Structural		$\mathsf{G3ip}_{\mathsf{W}}$	$\mathsf{G3ip}_{+inv}$
I	$\vee_L$	$\vee_{R_i}$	$\wedge_L$	$\wedge_R$	$\top_R$	$\top_L$	$\bot_L$	$\supset_L$	$\supset_R$	$\supset_L^{pR}$	W	C	$\supset_R$	C
$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{F}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{F}}$	$\sim_{\mathtt{DN}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\sim_{\mathtt{DN}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$

The non-invertible rules in this system are $\vee_{R_i}$ and $\supset_L$. Note that $\supset_R$ is invertible but the implementation failed to prove it. The reason is that the proof for this case requires admissibility of W. More precisely, consider a derivation of the sequent $\Gamma, A \supset B \vdash F \supset G$ and suppose that the last applied rule was

$$\frac{\Gamma, A \supset B \vdash A \quad \Gamma, B \vdash F \supset G}{\Gamma, A \supset B \vdash F \supset G} \supset_L$$

By inductive hypothesis on the right premise, $\Gamma, B, F \vdash G$ is derivable. Considering the left premise, since $\Gamma, A \supset B \vdash A$ is derivable, admissibility of weakening implies that $\Gamma, A \supset B, F \vdash A$ is also derivable, hence $\Gamma, A \supset B, F \vdash G$ is derivable and the result follows. It turns out that the admissibility of W is automatically proved by the algorithms. Let $\mathsf{G3ip}_{\mathsf{W}}$ denote the system G3ip with the admissible rule W added: in this system, the invertibility of $\supset_R$ can be automatically proved.

Although the rule $\supset_L$ is not invertible, it is invertible in its *right premise*. That is, if $\Gamma, F \supset G \vdash C$ is derivable, then so is $\Gamma, G \vdash C$. This result can also be proved by induction on the height of the derivation and the implementation returns a positive answer (this corresponds to the entry $\supset_L^{pR}$ in the table above).

Finally, as mentioned in Sect. 2, the proof of admissibility of contraction often requires the invertibility of rules. As an example, consider the derivation

$$\frac{\Gamma, F \supset G, F \supset G \vdash F \quad \Gamma, G, F \supset G \vdash C}{\Gamma, F \supset G, F \supset G \vdash C} \supset_L$$

By inductive hypothesis on the left premise, $\Gamma, F \supset G \vdash F$ is derivable and by invertibility of $\supset_L$ on the right premise, $\Gamma, G, G \vdash C$ is derivable and the result follows. Hence, by adding all the invertibilities already proved (system $\mathsf{G3ip}_{+inv}$ in the table), the tool was able to prove admissibility of the rule C.

As shown in Sect. 2, the proof of permutability of rules requires the invertibility lemmas and admissibility of weakening (already proved). Using the system $\mathsf{G3ip}_{+inv}$, the tool was able to prove all the permutability lemmas for propositional intuitionistic logic. The following table summarizes some of these results.

$\wedge_R \downarrow \wedge_L$	$\wedge_L \downarrow \wedge_R$	$\vee_i \downarrow \wedge_L$	$\wedge_L \downarrow \vee_i$	$\vee_{R_i} \downarrow \vee_L$	$\vee_L \downarrow \vee_{R_i}$	$\vee_{R_i} \downarrow \supset_L$	$\supset_L \downarrow \vee_{R_i}$	$\supset_L \downarrow \supset_L$	$\wedge_L \downarrow \supset_R$	$\supset_R \downarrow \wedge_L$
$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{F}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$

Note that the approach followed for G3ip, $\mathsf{G3ip}_{\mathsf{W}}$ and $\mathsf{G3ip}_{+inv}$ in this section provides an example of a modular proof, where theorems are added as hypothesis to the system. In this way, more involved properties can be discarded.

6.2 Multi-conclusion Propositional Intuitionistic Logic (mLJ)

Maehara's mLJ [13] is a multiple conclusion system for intuitionistic logic. The rules are exactly the same as in G3ip, except for the $\vee_R$ and implication (see Fig. 2). While the left rule copies the implication in the left premise, the right implication forces all formulas in the succedent of the conclusion sequent to be weakened (when viewed bottom-up). This guarantees that, when the $\supset_R$ rule is applied on $A \supset B$, the formula B should be proved assuming *only* the pre-existent antecedent context extended with the formula A. This creates an interdependency between A and B.

$$\frac{\Gamma, A \supset B \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \supset B \vdash \Delta} \supset_L \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B, \Delta} \supset_R \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee_R$$

Fig. 2. The multi-conclusion intuitionistic sequent system mLJ.

The introduction rules in mLJ are invertible, with the exception of $\supset_R$. In particular, two different applications of $\supset_R$ (on the same sequent) do not permute. For instance, from the premise of

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B, C \supset D, \Delta} \supset_R$$

the sequent $\Gamma, C \vdash D$ is not derivable. The results for this system are summarized in the table below:

Invertibilities										Structural		mLJ_{+inv}
I	$\vee_L$	$\vee_R$	$\wedge_L$	$\wedge_R$	$\top_R$	$\top_L$	$\bot_L$	$\supset_L$	$\supset_R$	W	C	C
$\surd_T$	$\surd_T$	$\surd_T$	$\surd_T$	$\surd_T$	$\surd_T$	$\surd_T$	$\surd_T$	$\surd_T$	$\surd_F$	$\surd_T$	$\sim_{DN}$	$\surd_T$

6.3 Propositional Classical Logic (G3cp)

G3cp [21] is a well known two-sided sequent system for classical logic, where the structural rules are implicit and all the rules are invertible. Differently from G3ip, weakening is not needed for the proof of invertibility of $\supset_R$. However, contraction still depends on invertibility results. The results are summarized below:

Assuming the already proved invertibility lemmas, the prover is able to show that, for all pair of rules r_1, r_2 in the system, $r_1 \downarrow r_2$.

Invertibilities										Structural		G3cp+*inv*
I	$\vee_L$	$\vee_R$	$\wedge_L$	$\wedge_R$	$\top_R$	$\top_L$	$\bot_L$	$\supset_L$	$\supset_R$	W	C	C
$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\sim_{\mathtt{DN}}$	$\surd_{\mathtt{T}}$

6.4 Linear Logic (LL)

Linear logic [8] is a resource-conscious logic, in the sense that formulas are consumed when used during proofs, unless they are marked with the exponential ? (whose dual is !), in which case, they behave *classically*. Propositional LL connectives include the additive conjunction & and disjunction $\oplus$ and their multiplicative versions $\otimes$ and $\parr$. The proof system for one-sided (classical) propositional linear logic is depicted in Fig. 3.

$$\frac{}{\vdash p^{\perp}, p}\ I \qquad \frac{}{\vdash 1}\ 1 \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \bot}\ \bot \qquad \frac{}{\vdash \Gamma, \top}\ \top$$

$$\frac{\vdash \Gamma_1, A \quad \vdash \Gamma_2, B}{\vdash \Gamma_1, \Gamma_2, A \otimes B}\ \otimes \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ \parr \qquad \frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \,\&\, B}\ \&$$

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B}\ \oplus_1 \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}\ \oplus_2 \qquad \frac{\vdash ?A_1, \ldots, ?A_n, A}{\vdash ?A_1, \ldots, ?A_n, !A}\ !$$

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, ?A}\ ? \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?A}\ W \qquad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}\ C$$

Fig. 3. One-sided monadic system LL.

$$\frac{\vdash \Theta, F : \Gamma}{\vdash \Theta : \Gamma, ?F}\ ? \qquad \frac{\vdash \Theta, F : \Gamma, F}{\vdash \Theta, F : \Gamma}\ copy \qquad \frac{\vdash \Theta : \Gamma_1, A \quad \vdash \Theta : \Gamma_2, B}{\vdash \Theta : \Gamma_1, \Gamma_2, A \otimes B}\ \otimes$$

Fig. 4. Some rules of the dyadic system D−LL.

Since formulas of the form $?F$ can be contracted and weakened, such formulas can be treated as in classical logic, while the remaining formulas are treated linearly. This is reflected into the syntax of the so called *dyadic sequents* (Fig. 4) which have two contexts: Θ is a set of formulas and Γ a multiset of formulas. The sequent $\vdash \Theta : \Gamma$ is interpreted as the linear logic sequent $\vdash ?\Theta, \Gamma$ where $?\Theta = \{?A \mid A \in \Theta\}$. It is then possible to define a proof system without explicit

weakening and contraction (system $\mathsf{D{-}LL}$ in Fig. 4). The complete dyadic proof system can be found in [1].

Since propositional LL is undecidable [12], infinite computations are possible. In this case study, a search bound is used to force termination of the implementation. Since all the theorems include a very controlled number of connectives (usually the 2 connectives involved in the application of the rules), this seems to be a fair solution.

For the monadic (LL) and the dyadic ($\mathsf{D{-}LL}$) systems, the results of invertibility of rules are summarized in the next table.

LL and D−LL								LL			D−LL		$\mathsf{D{-}LL}_{+\mathsf{W}_c}$
1	$\bot$	$\top$	$\otimes$	$\&$	$⅋$	$\oplus_i$	$!$	$?$	$?_{\mathsf{C}}$	$?_{\mathsf{W}}$	$?$	*copy*	$?$
$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{F}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{F}}$	$\surd_{\mathtt{F}}$	$\surd_{\mathtt{F}}$	$\surd_{\mathtt{T}}$	$\surd_{\mathtt{F}}$	$\sim_{\mathtt{DN}}$	$\surd_{\mathtt{F}}$	$\surd_{\mathtt{T}}$

In LL, the rules $?$ (dereliction) and $?_{\mathsf{W}}$ (weakening) are not invertible, while $?_{\mathsf{C}}$ (contraction) is invertible. In $\mathsf{D{-}LL}$, the rule $?$ is invertible. However, the proof of this theorem fails for the case $\otimes$. To obtain a proof, first admissibility of weakening for the classical context is proved: if $\vdash \Theta : \Gamma$ is derivable, then $\vdash \Theta, \Theta' : \Gamma$ is derivable (rule W_c). $?$ is proved invertible in $\mathsf{D{-}LL}_{+\mathsf{W}_c}$.

Finally, the prover was able to discharge the following theorems:

– (LL) If $\vdash \Gamma, !F$ is derivable then $\vdash \Gamma, F$ is derivable.
– ($\mathsf{D{-}LL}$) If $\vdash \Theta : \Gamma, !F$ is derivable then $\vdash \Theta : \Gamma, F$ is derivable.

6.5 Normal Modal Logics: K and S4

A modal is an expression (like *necessarily* or *possibly*) that is used to qualify the truth of a judgment, e.g., $\Box A$ can be read as "the formula A is necessarily true". The most familiar modal logics are constructed from the modal logic K and its extensions are called *normal modal logics*. The system $\mathsf{S4}$ is an extension of K where $\Box\Box A \equiv \Box A$ holds. Figure 5 presents the modal sequent rules for K and $\mathsf{S4}$.

$$\frac{\Gamma \vdash A}{\Gamma', \Box\Gamma \vdash \Box A, \Delta}\ \mathsf{k} \qquad \frac{\Gamma, \Box A, A \vdash \Delta}{\Gamma, \Box A \vdash \Delta}\ \mathsf{T} \qquad \frac{\Box\Gamma \vdash A}{\Gamma', \Box\Gamma \vdash \Box A, \Delta}\ 4$$

Fig. 5. The modal sequent rules for K (k) and $\mathsf{S4}$ ($\mathsf{k} + \mathsf{T} + 4$)

All the propositional rules are invertible in both K and $\mathsf{S4}$, k and 4 are not invertible (due to the implicit weakening) while T is invertible. Similar to the previous systems, the admissibility of W follows immediately and the proof of admissibility of C requires as hypotheses the already proved invertibility lemmas:

Invertibilities										Structural		Modal Rules			K_{+inv}	$\mathsf{S4}_{+inv}$
I	$\vee_L$	$\vee_R$	$\wedge_L$	$\wedge_R$	$\top_R$	$\top_L$	$\bot_L$	$\supset_L$	$\supset_R$	W	C	k	4	T	C	C
$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\sim_{\mathtt{DN}}$	$\checkmark_{\mathtt{F}}$	$\checkmark_{\mathtt{F}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$	$\checkmark_{\mathtt{T}}$

7 Related Work and Concluding Remarks

The proposal of many proof systems for many logics demanded trustful methods for determining good properties. In general, the checking was normally done via a case-by-case analysis, by trying exhaustively all the possible combinations of application of rules in a system. The advent of automated reasoning changed completely the scenery, since theorems started being proved automatically in meta-level frameworks. This has brought a whole new perspective to the field of proof theory: useless proof search steps, usually singular to a specific logic, have been disregarded in favor of developing general and universal methods for providing good automation strategies. These developments have ultimately helped in determining general conceptual characteristics of logical systems, as well as in identifying effective meta-level frameworks that can capture (and reason about) them in a natural way.

This work moves forward towards this direction: it proposes a general, natural, and uniform way of proving key properties of sequent systems using the rewriting logic framework, enabling modular proofs of meta-level properties of logical systems. Permutability of rules is a nice start case study since it is heavily used in cut-elimination proofs. Moreover, permutability has a rewriting counterpart: showing that applying a rule r_1 followed by a rule r_2 is the same as applying r_2 then r_1 can be interpreted as having the diamond property on the application of these two rules. The proof of permutability itself does not need inductive methods explicitly: they are hidden in other needed results like admissibility of weakening and invertibility of rules. The approach adopted in this work profits, as much as possible, from modularity. First permutability is tested without any other assumptions; then, if possible, prove admissibility of weakening and invertibility theorems; finally, add the proven theorems modularly to the system and re-run the permutability test: some cases for which the tool previously failed can now be proved. The same core algorithm can be used for proving admissibility of contraction, for example, which also depends on invertibility results.

The choice of rewriting logic as a meta-level framework brought advantages over some other options in the literature. Indeed, while approaches using logical frameworks depend heavily on the specification method and/or the implicit properties of the meta and object logics, rewriting logic enables the specification of the rules as they are actually written in text and figures. Consider for example the LF framework [20], based on intuitionistic logic, where the left context is handled by the framework as a set. Specifying sequent systems based on multisets requires elaborated mechanisms, which makes the encoding far from being natural. Moving from intuitionistic to linear logic solves this problem [4,16], but still several sequent systems cannot be naturally specified in the LL framework, as it is the case of mLJ. This latter situation can be partially fixed by adding

subexponentials to linear logic (SELL) [18,19], but then the resulting encoding although natural, is often non-trivial and it cannot be fully automated. Moreover, several logical systems cannot be naturally specified in SELL, such as K. All in all, this paper is yet another proof that rewriting logic is an innovative and elegant framework for reasoning about logical systems, since results and systems themselves can be modularly extended. In fact, the approach here can be extended to reason about a large class of systems, including normal (multi)-modal [11] and paraconsistent [9] sequent systems. The authors conjecture that the same approach can be used for extensions of sequent systems themselves, like nested [3] or linear nested [10] systems. This is an interesting future research path worth pursuing.

Finally, a word about cut-elimination. The usual cut-elimination proof strategy can be summarized by the following steps: (i) transforming a proof with cuts into a proof with principal cuts; (ii) transforming a proof with principal cuts into a proof with atomic cuts; (iii) transforming a proof with atomic cuts into a cut-free proof. While step (ii) is not difficult to solve (see e.g., [16]), steps (i) and (iii) strongly depend on the ability of showing *permutability* of rules. With the results presented in this work, it seems reasonable to envisage the use of the approach – both the techniques and their implementation – in order to fully automate cut-elimination proofs for various proof systems. It is worth noticing, though, that the aim of this paper is more general: proving results in a modular way permits maximizing their subsequent use in other applications as well. For example, it would be interesting to investigate further the role of invertible rules as equational rules in rewriting systems. While this idea sounds more than reasonable, it is necessary to check whether promoting invertible rules to equations preserves completeness of the system (e.g., the resulting equational theory needs to be, at least, ground confluent and terminating). If the answer to this question is in the affirmative for a large class of systems, then the approach presented here also opens the possibility, e.g., to automatically access focused systems [1].

Acknowledgments. The authors would like to thank the anonymous reviewers for their valuable comments on an earlier draft of this paper. The work of the three authors was supported by CAPES, Colciencias, and INRIA via the STIC AmSud project “EPIC: EPistemic Interactive Concurrency” (Proc. No 88881.117603/2016-01). The work of Pimentel and Olarte was also supported by CNPq and the project FWF START Y544-N23.

References

1. Andreoli, J.-M.: Logic programming with focusing proofs in linear logic. J. Logic Comput. **2**(3), 297–347 (1992)
2. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoret. Comput. Sci. **360**(1–3), 386–414 (2006)
3. Brünnler, K.: Deep sequent systems for modal logic. Arch. Math. Logic **48**, 551–577 (2009)

4. Cervesato, I., Pfenning, F.: A linear logical framework. Inf. Comput. **179**(1), 19–75 (2002)
5. Ciabattoni, A., Galatos, N., Terui, K.: From axioms to analytic rules in nonclassical logics. In: LICS, pp. 229–240. IEEE Computer Society Press (2008)
6. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
7. Gentzen, G.: Investigations into logical deduction. In: Szabo, M.E. (ed.) The Collected Papers of Gerhard Gentzen, North-Holland, pp. 68–131 (1969)
8. Girard, J.-Y.: Linear logic. Theoret. Comput. Sci. **50**, 1–102 (1987)
9. Lahav, O., Marcos, J., Zohar, Y.: Sequent systems for negative modalities. Logica Universalis **11**(3), 345–382 (2017)
10. Lellmann, B.: Linear nested sequents, 2-sequents and hypersequents. In: De Nivelle, H. (ed.) TABLEAUX 2015. LNCS (LNAI), vol. 9323, pp. 135–150. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24312-2_10
11. Lellmann, B., Pimentel, E.: Proof search in nested sequent calculi. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 558–574. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_39
12. Lincoln, P., Mitchell, J., Scedrov, A., Shankar, N.: Decision problems for propositional linear logic. Ann. Pure Appl. Logic **56**, 239–311 (1992)
13. Maehara, S.: Eine darstellung der intuitionistischen logik in der klassischen. Nagoya Math. J. **7**, 45–64 (1954)
14. Martí-Oliet, N., Meseguer, J.: Rewriting logic as a logical and semantic framework. In: Gabbay, D.M., Guenthner, F. (eds.) Handbook of Philosophical Logic, pp. 1–87. Springer, Dordrecht (2002)
15. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoret. Comput. Sci. **96**(1), 73–155 (1992)
16. Miller, D., Pimentel, E.: A formal framework for specifying sequent calculus proof systems. Theoret. Comput. Sci. **474**, 98–116 (2013)
17. Miller, D., Saurin, A.: From proofs to focused proofs: a modular proof of focalization in linear logic. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 405–419. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74915-8_31
18. Nigam, V., Pimentel, E., Reis, G.: An extended framework for specifying and reasoning about proof systems. J. Logic Comput. **26**(2), 539–576 (2016)
19. Nigam, V., Reis, G., Lima, L.: Quati: an automated tool for proving permutation lemmas. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 255–261. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_18
20. Pfenning, F.: Structural cut elimination I. Intuitionistic and classical logic. Inf. Comput. **157**(1/2), 84–141 (2000)
21. Troelstra, A.S., Schwichtenberg, H.: Basic Proof Theory. Cambridge University Press, New York (1996)
22. Viry, P.: Equational rules for rewriting logic. Theoret. Comput. Sci. **285**(2), 487–517 (2002)

Formal Modeling and Analysis of the Walter Transactional Data Store

Si Liu[1(✉)], Peter Csaba Ölveczky[2], Qi Wang[1], and José Meseguer[1]

[1] University of Illinois, Urbana-Champaign, USA
siliu3@illinois.edu
[2] University of Oslo, Oslo, Norway

Abstract. Walter is a distributed partially replicated data store providing Parallel Snapshot Isolation (PSI), an important consistency property that offers attractive performance while ensuring adequate guarantees for certain kinds of applications. In this work we formally model Walter's design in Maude and formally specify and verify PSI by model checking. To also analyze Walter's performance we extend the Maude specification of Walter to a probabilistic rewrite theory and perform statistical model checking analysis to evaluate Walter's throughput for a wide range of workloads. Our performance results are consistent with a previous experimental evaluation and throw new light on Walter's performance for different workloads not evaluated before.

1 Introduction

Cloud-based transaction systems provide both a challenge and an opportunity for the use of formal methods. The *challenge* has to do with the fact that the very *raison d'être* for such system is the need for a carefully chosen compromise between consistency guarantees and performance. Their massive use requires them to ensure scalability to large numbers of users with acceptable latency and throughput, while also guaranteeing the promised consistency properties. This is a challenge for formally-based design, because many formal methods tend to solely focus on correctness. Yet, correctness without due performance is useless for these systems. The *opportunities* are plentiful, including the following: (1) Many of these systems have never been formally specified, either at the *system specification* level or at the *property specification* level. (2) There is a need for *modularity* and *conceptual unification* in the design of these, currently quite ad-hoc and monolithic, systems. (3) There is also the prospect of using formal executable specifications for *code generation* purposes, achieving correct-by-construction systems that, by having been thoroughly analyzed in their correctness and performance aspects, can achieve very high quality.

This work is part of a long-term research effort in which we have been using Maude to meet the challenges and exploit the opportunities described above for cloud-based transaction systems (see [4] for a survey). Specifically, we exploit the type-(1) opportunity offered by Walter [21], a well-known cloud-based transaction system that provides an important intermediate consistency

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 136–152, 2018.
https://doi.org/10.1007/978-3-319-99840-4_8

guarantee, Parallel Snapshot Isolation (PSI). Walter is a very good type-(1) opportunity because no formal system specification exists at all; and there is no formal (or even informal) verification that it guarantees PSI. Walter is also a good stepping stone towards placing the design of cloud-based transaction systems in a formally-based modular framework. The way we are advancing this type-(2) goal is by first systematically studying system designs that cover the whole spectrum between lower-guarantees/higher-performance and higher-guarantees/lower-performance systems. We have already studied several systems in this spectrum, including RAMP [11,15], our own ROLA design [12], P-Store [18], and Megastore [9]. Walter has been a key missing design in the spectrum. The essential point is that case studies spanning the entire correctness/performance spectrum are crucial for identifying optimal decompositions of such systems into modular, reusable components. Finally, the fact that our Maude specification of Walter and of the other above-mention systems are *executable*, also helps us advance towards exploiting the type-(3) opportunity of achieving high-quality code generation for formal specifications. In this paper we focus on the type-(1) goal for Walter, but our sights are aimed at the type-(2)–(3) goals just as much.

Main Contributions and Outline. In Section 2 we give an overview of Walter, the PSI property and the stronger Snapshot Isolation (SI) property, and summarize the main features of Maude used in this paper. Section 3 provides a formal executable specification of Walter in Maude. This is a key contribution since, to the best of our knowledge, it is the first formal specification of Walter. Section 4 formalizes the SI and PSI properties and formally analyzes for the first time whether the Walter design satisfies either of these properties. This analysis is achieved by: (i) providing a parametric method to generate all initial states for given parameters; and (ii) performing model checking analysis to verify the SI and PSI properties for all initial states for various parameter choices. To analyze complex properties such as SI and PSI we propose a new general method for model checking such properties by adding a "monitor" to the state which records the global order of transaction starts and commits/aborts. In this way we can easily specify and model check SI and PSI; furthermore, this technique should also be applicable to analyze other consistency properties. Our analysis shows that the Walter design does indeed satisfy the PSI property for all our initial states but fails to satisfy the SI property. Section 5 makes four contributions. First, it extends the Maude model of Walter from a rewrite theory to a *probabilistic* rewrite theory by adding time and probability distributions for message delays to the original specification. Second, it carries out a systematic *statistical model checking* analysis of the key performance metric, transaction throughput, under a wide range of workloads. Third, it confirms that the performance estimates thus obtained are consistent with those obtained experimentally for the Walter implementation in [21]. Fourth, it provides new insights about Walter's performance beyond the limited ranges for which such information was available by experimental evaluation in [21]. Finally, related work is discussed in Section 6, and concluding remarks are given in Section 7.

2 Preliminaries

Parallel Snapshot Isolation. To deal with huge amounts of data, cloud-based applications need to *partition* their data across distributed sites, and to provide high availability and disaster tolerance, data must be *replicated* at widely distributed sites. Such *partially replicated* data stores have to: (i) maintain some consistency of replicated data, and (ii) provide some consistency for (multi-partition) transactions that access data stored at different partitions. However, ensuring high degrees of consistency for partially replicated data stores supporting multi-partition transactions requires a lot of costly coordination, which might lead to unacceptable delays and throughput for many kinds of applications. Designers of distributed data stores therefore face a trade-off between providing good consistency properties and high performance. There are a number of consistency properties, ranging from strong ones like serializability all the way to weak properties such as read atomicity and eventual consistency.

A popular intermediate consistency model provided by commercial database systems such as Oracle and SQL Server is *snapshot isolation* (SI) [3]. The idea is that a multi-partition transaction reads from a *snapshot* of a distributed data store that reflects a *single commit order* of transactions across sites.

In [21], the authors argue that having the same commit order across all sites is not necessary for social networks and similar applications: it does not matter much that Vlad in Moscow sees Kim's post before seeing Benny's post, whereas Don in Washington sees Benny's post before Kim's post. (Hence Benny and Kim can commit their (independent) posts without waiting for each other.) They propose a new consistency model, called *parallel snapshot isolation*, which allows different commit orders at different sites, while still guaranteeing:

- recent and "consistent" views: all operations in a transaction read the most recent version committed at the transaction execution site, as of the time when the transaction begins;
- no write-write conflicts (the write sets of committed somewhere-concurrent transactions must be disjoint); and
- preservation of *causality* across sites, which ensures that both Vlad and Benny see Kim's post before seeing Don's reply to Kim's post.

In [21] the authors specify PSI by giving an abstract pseudo-code "program" of a centralized execution that a distributed implementation must emulate.

Walter. *Walter* [21] is a partially replicated geo-distributed data store that supports multi-partition transactions and guarantees/implements PSI.

The key idea to ensure that all operations in a transaction read a consistent "snapshot" of the distributed data store is that each site s maintains a (local) *vector timestamp* $\{site_1 \mapsto k_1, \ldots, site_n \mapsto k_n\}$ representing a current snapshot of the state, as seen by site s, where $site_j \mapsto k_j$ means that the snapshot includes the first k_j transactions executed at site $site_i$. Each time a transaction starts executing at s, the transaction is assigned the current local snapshot/vector timestamp of site s. Remote reads can then be performed consistently according to

this snapshot. Another key Walter feature is that each data item has a preferred site, so that writes at preferred sites can be committed fast.

A transaction is executed as follows. When the "host" site s starts executing the transaction t, t is assigned the current snapshot of s. The site s then executes the read and write operations in t. For writes, Walter buffers the versions written in the transaction's write set. For reads, Walter fetches the latest appropriate version according to t's start snapshot, by checking any updates in the write set and its history of previous updates. If the data item is not replicated locally, Walter retrieves the right version remotely from the data item's preferred site.

When the host site has executed all the operations in a transaction, it starts committing the transaction. Read-only transactions and transactions that only write data items whose preferred site is the host site can commit locally (*fast commit*). Walter then checks whether all versions of each data item in the history of the local site are *unmodified* since the start vector timestamp, and whether all data items are *unlocked* (i.e., not being committed by another transaction). If either check fails, Walter aborts the transaction; otherwise, it can be committed. If a transaction t cannot commit locally (*slow commit*), the executing site s uses the two-phase commit (2PC) protocol to check whether t can be committed, by asking the preferred sites of data items written by t whether t can be committed. If the data items written by t are unmodified and unlocked at such a site, the site replies with a "yes" vote and locks the corresponding data items. Otherwise, the site votes "no." If the executing site receives a "no" vote, t is aborted and the other preferred sites are notified and release the appropriate locks. If all votes are "yes" votes, the transaction can be committed.

If the transaction t can be (fast or slow) committed, the site s marks t as committed, assigns it a *version* $(s, seqNo)$ (where *seqNo* is a local sequence number), updates the local history with the updates, and propagates t to other sites, which update their histories and their vector timestamps. To allow f site failures, a transaction is marked *disaster-safe durable* if its writes have been logged at $f+1$ sites. The propagation protocol first checks whether the transaction can be marked as disaster-safe durable by collecting acknowledgments from $f + 1$ sites for each data item. Upon receiving the propagation of a transaction, a site acknowledges it only after it receives all transactions that causally precede the propagated transaction, and all transactions at the same executing site with a smaller sequence number. The protocol then checks whether the transaction can be marked as globally visible. This is done by committing the transaction at all sites. A transaction can be committed at a remote site when it learns that the transaction is disaster-safe durable, all transactions causally preceding the transaction have been committed locally, and all transactions at the same executing site with a smaller sequence number have been committed locally.

The paper [21] briefly discusses failure handling, but does not give much detail. The authors have implemented Walter in about 30K lines of code, and have implemented Facebook- and Twitter-like applications on top of Walter using the Amazon EC2 cloud platform to evaluate Walter's performance in a distributed setting (with nodes in US, Ireland, and Singapore). They use their distributed deployment

to estimate the transaction latency and throughput (committed transactions per second) for read-only, write-only, and 90% read workloads.

The authors to do not prove or justify that Walter actually guarantees PSI.

Rewriting Logic and Maude. In rewriting logic a concurrent system is specified a as *rewrite theory* $(\Sigma, E \cup A, R)$, where $(\Sigma, E \cup A)$ is a *membership equational logic theory* [6], with Σ an algebraic signature declaring sorts, subsorts, and function symbols, E a set of conditional equations, and A a set of equational axioms. It specifies the system's state space as an algebraic data type. R is a set of *labeled conditional rewrite rules*, specifying the system's local transitions, of the form $[l] : t \longrightarrow t'$ **if** *cond*, where *cond* is a condition and l is a label. Such a rule specifies a transition from an instance of t to the corresponding instance of t', provided the condition holds.

Maude [6] is a language and tool for specifying, simulating, and model checking rewrite theories. The distributed state of an object-oriented system is formalized as a *multiset* of objects and messages. A class C with attributes att_1 to att_n of sorts s_1 to s_n is declared `class` C | $att_1 : s_1, \ldots, att_n : s_n$. An object of class C is modeled as a term `<` $o : C$ | $att_1 : v_1, \ldots, att_n : v_n$ `>`, with o its object identifier, and where the attributes att_1 to att_n have the current values v_1 to v_n, respectively. Upon receiving a message, an object can change its state and/or send messages to other objects. For example, the rewrite rule

```
rl [l] :  m(O,z)  < O : C | a1 : x, a2 : O' >
       =>         < O : C | a1 : x + z, a2 : O' >  m'(O',x + z) .
```

defines a transition where an incoming message `m`, with parameters `O` and `z`, is consumed by the target object `O` of class `C`, the attribute `a1` is updated to `x + z`, and an outgoing message `m'(O',x + z)` is generated.

Statistical Model Checking and PVeStA. Probabilistic distributed systems can be modeled as *probabilistic rewrite theories* [1] with rules of the form

$$[l] : t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \;\; \textbf{if} \;\; cond(\overrightarrow{x}) \;\; \textit{with probability} \;\; \overrightarrow{y} := \pi(\overrightarrow{x})$$

where the term t' has new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ in the term t. The concrete values of the new variables $\overrightarrow{y}$ in $t'(\overrightarrow{x}, \overrightarrow{y})$ are chosen probabilistically according to the probability distribution $\pi(\overrightarrow{x})$.

Statistical model checking [19,22] is an attractive formal approach to analyzing (purely) probabilistic systems. Instead of offering a yes/no answer, it can verify a property up to a user-specified level of confidence by running Monte-Carlo simulations of the system model. We then use PVeStA [2], a parallelization of the tool VeStA [20], to statistically model check purely probabilistic systems against properties expressed as QuaTEx expressions [1]. The expected value of a QuaTEx expression is iteratively evaluated w.r.t. two parameters α and δ by sampling, until we obtain a value v so that with $(1-\alpha)100\%$ statistical confidence, the expected value is in the interval $[v - \frac{\delta}{2}, v + \frac{\delta}{2}]$.

3 A Formal Model of Walter in Maude

This section explains how we have formalized Walter in Maude. Due to space limitations, we only show parts of our model (e.g., 4 out of 26 rewrite rules) and refer to the accompanying longer report [13] and the executable model available at https://sites.google.com/site/siliunobi/walter for more details.

We formalize Walter in an object-oriented style, where the state consists of a number of *replica* (or *site*) objects, each modeling a local database, and a number of messages traveling between the objects. A *transaction* is formalized as an object which resides inside the replica object that executes the transaction.

Some Data Types. A *version* is a pair `version(oid,sqn)` consisting of a site `oid` where the transaction is executed, and a sequence number `sqn` local to that site. A vector timestamp is a map from site identifiers to sequence numbers. The sort `OperationList` represents lists of read and write operations as terms such as `(`x `:= read` $k1$`) (`y `:= read` $k2$`) write(`$k1, x+y$`)`, where x and y are "local variables." An "operation" `waitRemote(`k, x`)` means that the transaction execution is awaiting the value of the key/data item k from a remote site to be assigned to the local variable x (see [13] for the definition of these data types).

Classes. A *transaction* is modeled as an object of the following class `Txn`:

```
class Txn | operations : OperationList,  readSet : ReadSet,
            writeSet : WriteSet,         localVars : LocalVars,
            startVTS : VectorTimestamp,  txnSQN : Nat .
```

The `operations` attribute denotes the transaction's remaining operations. The `readSet` attribute denotes the versions of data items read by the transaction as a ','-separated set of pairs `versionRead(`k, *version*`)`. `writeSet` denotes the write set of the transaction as a map `(`k_1 `|->` val_1`), ..., (`k_n `|->` val_n`)`. `localVars` maps the transaction's local variables to their values. `startVTS` is the vector timestamp assigned to the transaction when it starts to execute, and `txnSQN` is the transaction's sequence number given upon commit.

A *replica*, or *site*, stores parts of the database and executes the transactions for which it is the host, and is formalized as an object of the following class:

```
class Replica | history : Datastore,  sqn : Nat,  gotTxns : ObjectList,
                executing : ObjectList,  committed : ObjectList,
                aborted : ObjectList,  committedVTS : VectorTimestamp,
                gotVTS : VectorTimestamp,  locked : Locks,
                votes : Vote,  voteSites : TxnSites,  abortSites : TxnSites,
                dsSites : PropagateSites,  vsbSites : TxnSites,
                dsTxns : OidSet,  gvTxns : OidSet,
                recPropTxns : PropagatedTxns, recDurableTxns : DurableTxns .
```

The `history` attribute represents the site's local database, as well as propagated updates also on data items not stored at the replica, as a map from keys to *lists* of updates `<` *value*, *version* `>`. The `sqn` attribute denotes the replica's current

local sequence number. The attributes `gotTxns`, `executing`, `committed` and `aborted` denote the transaction (objects) which are, respectively, waiting to be executed, executing, committed, and aborted. The attributes `committedVTS` and `gotVTS` indicate for each site how many transactions of that site have been committed at, respectively, received by, this site. `locked` denotes the locked keys and their associated transactions at this site. The attributes `recPropTxns` and `recDurableTxns` buffer the received propagation and disaster-safe durable messages from the coordinator. See [13] for an explanation of the other attributes.

The state also contains an object mapping each key to the sites storing the key (these sites are also called the *replicas* of the key):

```
class Table | table : ReplicaTable .
```

Messages between sites have the form `msg` *content* `from` *sender* `to` *receiver*. We only introduce the messages appearing in the rewrite rules shown in this paper. The message content (or simply message) `request(`*key*, *txn*, *vts*`)` sends a read request for transaction `txn` to *key*'s preferred site to retrieve its state from the snapshot determined by vector timestamp *vts*. The preferred site replies with a message `reply(`*txn*, *key*, *value_version*`)`, where *value_version* is chosen based on the incoming vector timestamp. The message `ds-durable(`*txn*`)` is sent to all sites once the transaction *txn* has been marked as disaster-safe durable. The sites then reply with a message `visible(`*txn*`)` to acknowledge the notification.

Formalizing Walter's Behavior. The following three rules show how the host site `RID` executes a read operation `X :=read K` in the currently executing transaction `TID` when the transaction has not already written data item `K` (`not $hasMapping(WS,K)`) and the site `RID` does not replicate data item `K` (`not localReplica(K,RID,REPLICA-TABLE)`). In this case, the site sends a `request` message (with the transaction's start vector timestamp `VTS`) to `K`'s preferred site (`preferredSite(...)`) to fetch the version. The "next operation" of the transaction changes to `waitRemote(K,X)`:[1]

```
crl [execute-read-remote] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Replica | executing :
          < TID : Txn | operations : ((X :=read K) OPS), writeSet : WS,
                        startVTS : VTS > >
  =>
    < TABLE : Table | >
    < RID : Replica | executing :
          < TID : Txn | operations : (waitRemote(K,X) OPS) > >
    (msg request(K,TID,VTS) from RID to preferredSite(K,REPLICA-TABLE))
  if (not $hasMapping(WS,K)) /\ (not localReplica(K,RID,REPLICA-TABLE)) .
```

The remote (preferred) site responds to this request by sending the snapshot-consistent value and version (`choose(VTS, DS[K])`) of the requested key:

[1] We do not give variable declarations, but follow the convention that variables are written in (all) capital letters.

```
rl [receive-remote-request] :
   (msg request(K,TID,VTS) from RID' to RID)
   < RID : Replica | history : DS >
 =>
   < RID : Replica | >
   (msg reply(TID,K,choose(VTS,DS[K])) from RID to RID') .
```

The executing site then merges the fetched value and version in the local history, and updates the read set and local variables:

```
rl [receive-remote-reply] :
   (msg reply(TID,K,< V,VERSION >) from RID' to RID)
   < RID : Replica | history : DS, executing :
         < TID : Txn | operations : (waitRemote(K,X) OPS), readSet : RS,
                       localVars : VARS > >
 =>
   < RID : Replica | executing :
         < TID : Txn | operations : OPS,
                       readSet : (versionRead(K,VERSION),RS),
                       localVars : insert(X,V,VARS) >,
                     history : merge(K,< V,VERSION >,DS) > .
```

The last rule we show concerns the propagation of committed transactions. If a transaction TID can be committed, it is propagated to the other sites. When a receiving site has received all transactions that causally precede TID and all transactions from TID's host site with smaller sequence numbers, the transaction TID is propagated successfully and this is acknowledged to the host site. When the host has received $f + 1$ such acknowledgments it marks TID as *disaster-safe durable* and sends a ds-durable message to each site. When a remote site receives this decision, the site tries to commit the transaction locally:

```
crl [receive-ds-durable-visible] :
    (msg ds-durable(TID) from RID' to RID)
    < RID : Replica | recPropTxns : (propagatedTxns(TID,SQN,VTS) ; PTXNS),
                      recDurableTxns : DTXNS, committedVTS : VTS',
                      locked : LOCKS >
  =>
    < RID : Replica | recDurableTxns : (durableTxns(TID) ; DTXNS),
                      committedVTS : insert(RID',SQN,VTS'),
                      locked : release(TID,LOCKS) >
    (msg visible(TID) from RID to RID')
    if VTS' gt VTS /\ s(VTS'[RID']) == SQN .
```

To commit transaction TID, the remote site must check that: (i) the propagation message has been received and acknowledged (propagatedTxns(TID,SQN,VTS) shown in recPropTxns); (ii) VTS' is greater than VTS; and (iii) all transactions from TID's host site with a smaller sequence number have been received (s(VTS'[RID']) == SQN). A visible message is then sent back, all corresponding locks are released, and the transaction is committed at the remote site.

4 Correctness Analysis

In this section we use reachability analysis—from all initial system configurations up to given bounds on the number of transactions, sites, etc.—to analyze whether Walter satisfies PSI and SI. To analyze these complex properties, we use a novel technique which adds a global "logical clock" to record the global order of transaction starts and commits/aborts.

4.1 Parametric Generation of Initial States

To analyze all possible initial configurations we introduce a new operator `init` so there is a one-step rewrite `init(`*parameters*`)` $\longrightarrow c_0$ for each possible initial configuration c_0. We declare a sort `ConfigSet` for *sets* of configurations, define a function `op initAux :` s_1 `...` s_n `-> ConfigSet` (see our report [13] for details) such that `initAux(`*params*,*params'*`)` generates all possible initial states for such parameters, and add the following rewrite rule to our model:

```
var C : Configuration .  var CS : ConfigSet .
crl [init] : init(params) => C if C ; CS := initAux(params,params') .
```

`init`'s parameters are the number of read-only transactions, the number of write-only transactions, the number of read-write transactions, the number of sites, the number of keys, and the replication factor. Each transaction has two operations.

One of 768 initial states generated by `init(1,1,1,2,2,2)` is

```
< 0 : Table | table : [replicatingSites(k1,1 2);;replicatingSites(k2,2 1)] >
< 1 : Replica |
      gotTxns : < 3 : Txn | operations : (k2l :=read k2) (k1l :=read k1),
                            readSet : empty, writeSet : empty,
                            localVars : (k1l |-> [0],k2l |-> [0]),
                            startVTS : empty, txnSQN : 0>,
      history : (k1 |-> < [0],version(0,0) >, k2 |-> < [0],version(0,0) >),
      sqn : 0,  ... >
< 2 : Replica |
      gotTxns : < 2 : Txn | operations : write(k2,1) write(k1,2), ... > ;;
                < 1 : Txn | operations : (k2l :=read k2) write(k2,1), ... >,
      history : k1 |-> < [0],version(0,0) >,k2 |-> <[0],version(0,0)>), ... >
```

where data items `k1` and `k2` are replicated at sites `1` and `2`, and have preferred sites `1` and 2, respectively. Site `1` has one read-only transaction '`3`'—with operations `(k2l :=read k2) (k1l :=read k1)`—to execute, and site 2 has one write-only transaction '`2`' and one read-write transaction '`1`' to execute.

4.2 Analyzing the Correctness Properties

This section analyzes whether Walter satisfies SI and PSI using a new technique where we record relevant history during a run. The SI and PSI properties can then be easily formalized as functions on the final state of an execution.

In particular, we add to the state an object

```
< m : Monitor | clock : clock, log : log >
```

which stores crucial information about the execution. The *clock* is a kind of "logical global clock" that totally orders transaction starts and commits/aborts. This clock is incremented by one whenever a transaction starts executing or is committed or aborted somewhere. The *log* maps each transaction to a record `record(`*rid*, *issueTime*, *finishTimes*, *committed*, *reads*, *writes*`)`, with *rid* its host site, *issueTime* its issue "time" according to the logical clock, *finishTimes* its commit/abort "time" at each site, *committed* a flag that is `true` if the transaction is committed, *reads* its key/versions read, and *writes* its write set.

We modify our rewrite rules to update the `Monitor` whenever a transaction starts or is committed/aborted somewhere. For example, when a site commits a propagated transaction, the monitor records the commit time `GT` for that transaction at site `RID` and increments the logical global time by one:

```
crl [receive-ds-durable-visible] :
    (msg ds-durable(TID) from RID' to RID)
    < M : Monitor | clock : GT,
                    log : (TID |-> record(RID',T1,VTS1,true,READS,WRITES)
                                  , LOG) >
    < RID : Replica | ... >    --- as before
  =>
    < M : Monitor | clock : GT + 1 ,
                    log : (TID |-> record(RID',T1,insert(RID,GT,VTS1) ,
                                                  true,READS,WRITES) , LOG) >
    < RID : Replica | ...  >   --- as before
    (msg visible(TID) from RID to RID')  if  ... .
```

Since Walter is terminating if a finite number of transactions are issued, we check the consistency properties by inspecting this monitor object in the final states, when all transactions have finished.

Formalizing Parallel Snapshot Isolation. As mentioned in Section 2, PSI is given by three properties [21]:

- PSI-1 (Site Snapshot Read): All operations read the most recent committed version at the transaction's site as of time when the transaction began.
- PSI-2 (No Write-Write Conflicts): The write sets of each pair of committed *somewhere-concurrent*[2] transactions must be disjoint.
- PSI-3 (Commit Causality Across Sites): If a transaction T_1 commits at a site A before a transaction T_2 starts at site A, then T_1 cannot commit after T_2 at any site.

[2] Two transactions are *somewhere-concurrent* if, at either host site, one of them has a commit timestamp between the start and the commit timestamp of the other.

We analyze PSI-2 (and all other properties) by searching for a reachable *final* state whose system log shows that the execution did not satisfy the property. The following function `p2-psi` checks whether PSI-2 holds in the execution reflected in the system log, by checking whether there is a write-write conflict between any pair of committed *somewhere-concurrent transactions* in the system log:

```
ops p1-psi p2-psi p3-psi : Log -> Bool .

ceq p2-psi(TID1 |-> record(RID1,TS1,(RID1 |-> TC ,VTS1),true,RS,
                           (v(X,V),WS)) ,
           TID2 |-> record(RID2,TS2,(RID1 |-> TC' ,VTS2),true,RS',
                           (v(X,V') ,WS')) , LOG) = false
    if TC' > TS1 and TC' < TC .

eq p2-psi(LOG) = true [owise] .
```

In the first equation there are two committed (their “commit” flags are `true`) transactions `TID1` and `TID2`—hosted, respectively, at sites `RID1` and `RID2`—that both wrote data item `X` (since `v(X, V)` and `v(X, V')` are in their respective write sets). `TID1` and `TID2` are somewhere-concurrent, since they are concurrent at `TID1`'s host site `RID1`: `TID2` committed at `RID1` at time `TC'`, which is between `TID1`'s start time `TS1` and its commit time `TC` at `RID1`.

The function `p3-psi` that analyzes PSI-3 by checking whether there was “bad situation” in which a transaction `TID1` committed at site `RID2` *before* a transaction `TID2` started at site `RID2` (`TC1 < TS2`), while `TID1` committed at site `RID` *after* `TID2` committed at site `RID` (`TC1 > TC2`):

```
ceq p3-psi((TID1 |-> record(RID1,TS1,(RID2 |-> TC ,RID |-> TC1 ,VTS1),
                            true,RS,WS),
            TID2 |-> record(RID2,TS2,(RID1 |-> TC' ,RID |-> TC2 ,VTS2),
                            true,RS',WS') , LOG)) = false
    if TC < TS2 /\ TC1 > TC2 .

eq p3-psi(LOG) = true [owise] .
```

We have equally easily defined a function `p1-psi` for property PSI-1, and functions `p1-si` and `p2-si` characterizing the executions where the requirements SI-1 and SI-2 for SI hold (see [13] for details).

We have analyzed Walter from *all* initial states with up to 3 transactions, 2 sites, 2 keys, and 2 replicas per key, as well as from a number of initial states with 3 transactions. The following command searches for a reachable final state where the log shows that PSI-2 is violated:

```
Maude> (search [1] init(1,0,2,2,2,2) =>!
          < M:Oid : Monitor | log : LOG:Log >  C:Configuration
             such that not p2-psi(LOG:Log) .)

No solution
```

Our analysis found that Walter may violate both SI-1 and SI-2, but did not uncover a violation of PSI. Each `search` command took about 2 h (worst-case) to execute on a 3.4 GHz × 8 Intel Core i7-2600 CPU with 11.7 GB memory.

5 Performance Estimation by Statistical Model Checking

In this section we use PVeStA statistical model checking to estimate the performance of Walter in a wider range of settings than in the experiments in [21], thereby providing further insight about Walter. For example, the experiments with fast commit in [21] assume full replication, whereas we also experiment with a partially replicated setting (which necessitates remote reads, etc.), and with workloads involving both slow and fast commits.

Probabilistic Modeling of Walter. For statistical model checking in PVeStA we need to eliminate nondeterminism in the untimed model in Section 3, and for performance estimation we need to add time and probabilities. All of this can be achieved by following the techniques in [8] and *probabilistically* assign to each message a *delay*. The idea is that if each rewrite rule is triggered by the arrival of a message (either directly, or indirectly by becoming enabled as a result of applying a rule that is triggered by the arrival of a message) and the message delay is sampled probabilistically from a dense time interval, then the probability that two messages have the same delay is 0, and hence no two actions are enabled simultaneously, eliminating nondeterminism and introducing time.

In more detail, nodes send messages of the form `[`Δ`,`*rcvr* `<-` *msg*`]`, where Δ is the message delay, *rcvr* is the recipient, and *msg* is the message content. When time Δ has elapsed, this message becomes a *ripe* message `{`T`,`*rcvr* `<-` *msg*`}`, where T is the "current global time" (used for analysis purposes only). Such a ripe message must then be consumed by the receiver *rcvr* before time advances.

We exemplify with the rule `receive-remote-request` how we have transformed the untimed non-probabilistic rewrite rules to the timed and probabilistic setting. In the probabilistic rule below, the incoming message `request` is equipped with the current global time `T`, and the outgoing message `reply` is equipped with a delay `D` sampled from the probability distribution `distr(...)`:

```
rl [receive-remote-request-prob] :
   {T, RID <- request(K,TID,VTS,RID')}
   < RID : Replica | history : DS >
 =>
   < RID : Replica | >
   [D, RID' <- reply(TID,K,choose(VTS,DS[K]),RID)]
  with probability D := distr(...) .
```

Extracting Performance Measures from Executions. This time we add to the state a monitor object `<` *m* `: Monitor | log:` *log* `>`. The logical clock is no longer needed, since now "real" time is given by the message arrival times. Furthermore, since we now analyze transaction throughput, the log is simpler: a list

of records `record(`*tid*, *issueTime*, *finishTime*, *committed*`)`, with *tid* the transaction identifier, *issueTime* its issue time, *finishTime* its commit/abort time, and *committed* a flag that is `true` if *tid* is committed.

We define a number of functions on (states with) such a monitor that extract different performance metrics from this "execution log." The function `throughput` computes the number of committed transactions per time unit:

```
op throughput : Config -> Float [frozen] .
eq throughput(< M : Monitor | log: LOG >  REST)
 = committedNumber(LOG) / totalRunTime(LOG) .
```

where `committedNumber` gives the number of committed transactions in `LOG` and `totalRunTime` returns the time when all transactions are finished (i.e., the largest *finishTime* in `LOG`) (see [13] for details).

Experimental Setup. We performed our experiments with 100 transactions, 1 or 5 operations per transaction, 100 keys, and up to 4 sites. The number of sites and the transaction size are the same as in the experiments in [21]. We used lognormal message delay distributions with parameters $\mu = 3$ and $\sigma = 1$ for local delays, and $\mu = 1$ and $\sigma = 2$ for remote delays.

Generating Initial States. Statistical model checking verifies a property up to a user-specified level of confidence by running Monte-Carlo simulations from a given initial state. We use an operator `probInit` to *probabilistically* generate initial states: `probInit(`*rtx*, *wtx*, *rwtx*, *sites*, *keys*, *rf*, *rops*, *wops*, *rwops*, *distr*`)` generates an initial state with *rtx* read-only transactions, *wtx* write-only transactions, *rwtx* read-write transactions, *sites* sites, *keys* keys, *rf* replication level, *rops* operations per read-only transaction, *wops* operations per write-only transaction, *rwops* operations per read-write transactions, and *distr* the key access distribution (the probability that an operation accesses a certain key). To capture the fact that some keys may be accessed more frequently than others, we also use Zipfian distributions in our experiments.

Each PVeStA simulation starts from `probInit`, which rewrites to a *different* initial state in each simulation. The reason is that this expression involves generating certain values—such as the transactions—*probabilistically.*

Statistical Model Checking Results. The plots in Fig. 1 show the *throughput* with only fast commit as a function of the number of sites, with read-only, write-only and 90% reads workload, and with uniform and Zipfian distributions. The plots show that read throughput scales nearly linearly with the number of sites; write throughput also grows with the number of sites, but not linearly. With a mixed workload, throughput is mostly determined by the transaction size. Our results are consistent with those in [21]. For uniform distribution we only plot the results with a mixed workload; for the other two workloads, the results are consistent with those using the Zipfian distribution.

The plots in Fig. 2 show the throughput with both fast and slow commit under the same experimental settings as in Fig. 1. As shown in the left plot,

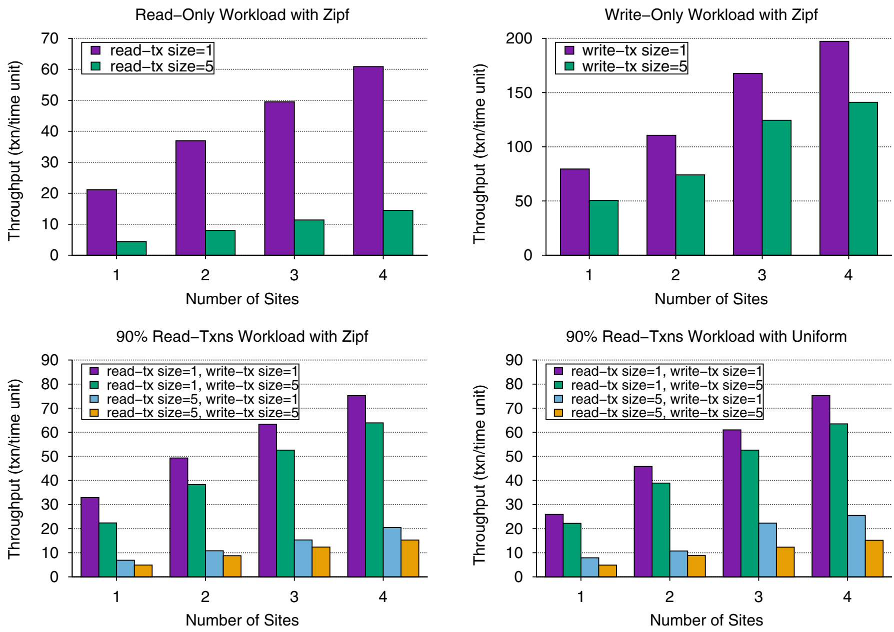


Fig. 1. Throughput with fast commit under different workloads.

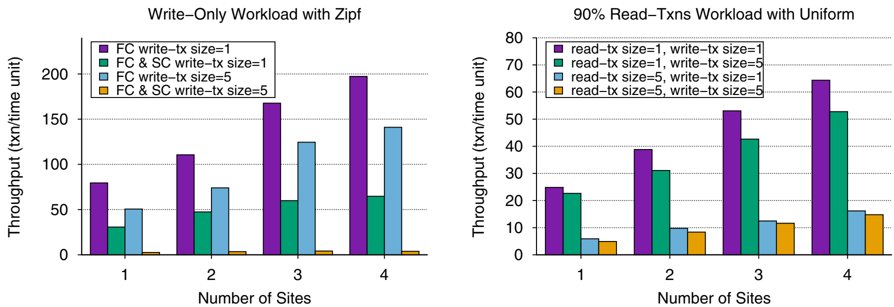


Fig. 2. Throughput with fast commit (FC) and slow commit (SC).

throughput is mostly determined by the transaction size in the mixed workload; the differences among various transaction sizes are consistent with those in Fig. 1.

Our Maude model, including the infrastructure for statistical model checking, is around 1.8K LOC. Computing the probabilities took a couple of minutes on 30 servers, each with a 64-bit Intel Quad Core Xeon E5530 CPU with 12 GB memory. Each point in the plots represents the average of 3 statistical model checking results. The confidence level for all our statistical experiments is 95%.

6 Related Work

Maude and PVeStA have been used to model and analyze the correctness and performance of a number of distributed data stores: the Cassandra key-value store [14,16], RAMP [11,15], Google's Megastore [9,10], and P-Store [18]. In contrast to these papers, our paper formalizes a different state-of-the-art algorithm, Walter, and, in particular, shows how the *snapshot isolation* and *parallel snapshot isolation* consistency models can be formalized and analyzed in Maude. In [12] we use PVeStA to compare the performance of our own new ROLA design with that of Walter. However, that paper focused on ROLA, and did not present the formalization of Walter or the SI and PSI properties.

In other applications of formal methods for distributed data stores, engineers at Amazon have used TLA+ and its model checker TLC to model and analyze the correctness of key parts of Amazon's celebrated cloud computing infrastructure [17]. In contrast to our work, they only use formal methods for correctness analysis. The designers of the TAPIR transaction protocol for distributed storage systems have also specified and model checked correctness (but not performance) properties of their design using TLA+ [23].

The papers [5,7] formalize a number of consistency models, including SI and PSI, but do not show how to analyze these properties.

7 Conclusions

We have formally analyzed and verified in Maude the design of Walter [21], a partially replicated distributed data store providing multi-partition transactions and guaranteeing parallel snapshot isolation (PSI), an important consistency property that offers attractive performance while providing adequate guarantees for certain kinds of applications. No formal specification of Walter existed before this work. Furthermore, PSI was only informally described by pseudo-code in [21] and no formal (or informal) verification existed. Using a logical clock to record the order of important events in an execution, we could use model checking and systematic generation of initial states to verify that Walter satisfies PSI for all such states. This technique should also make it easy to formalize and model check other consistency properties. We have also extended the Maude specification of Walter to model time and probabilistic communication delays as a probabilistic rewrite theory, and have then used statistical model checking analysis to study Walter's performance for a wide range of workloads. The results of the statistical model checking analysis are consistent with the experimental results in [21] but offer also new insights about Walter's performance for a wider range of workloads than those evaluated in [21].

Acknowledgments. We thank the anonymous reviewers for helpful comments on a previous version of this paper. This work has been partially supported by NSF Grant CNS 14-09416 and NRL under contract number N00173-17-1-G002.

References

1. Agha, G.A., Meseguer, J., Sen, K.: PMaude: rewrite-based specification language for probabilistic object systems. Electr. Notes Theor. Comput. Sci. **153**(2), 213–239 (2006)
2. AlTurki, M., Meseguer, J.: PVeStA: a parallel statistical model checking and quantitative analysis tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22944-2_28
3. Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: SIGMOD 1995. ACM (1995)
4. Bobba, R., et al.: Design, formal modeling, and validation of cloud storage systems using Maude. Technical report, University of Illinois at Urbana-Champaign (2017). http://hdl.handle.net/2142/96274
5. Cerone, A., Bernardi, G., Gotsman, A.: A framework for transactional consistency models with atomic visibility. In: CONCUR 2015. LIPIcs, vol. 42 (2015)
6. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
7. Crooks, N., Pu, Y., Alvisi, L., Clement, A.: Seeing is believing: a client-centric specification of database isolation. In: PODC 2017. ACM (2017)
8. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Statistical model checking for composite actor systems. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 143–160. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37635-1_9
9. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's Megastore in Real-Time Maude. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 494–519. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54624-2_25
10. Grov, J., Ölveczky, P.C.: Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 159–174. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_12
11. Liu, S., Ölveczky, P.C., Ganhotra, J., Gupta, I., Meseguer, J.: Exploring design alternatives for RAMP transactions through statistical model checking. In: Duan, Z., Ong, L. (eds.) ICFEM 2017. LNCS, vol. 10610, pp. 298–314. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_18
12. Liu, S., Ölveczky, P.C., Santhanam, K., Wang, Q., Gupta, I., Meseguer, J.: ROLA: a new distributed transaction protocol and its formal analysis. In: Russo, A., Schürr, A. (eds.) FASE 2018. LNCS, vol. 10802, pp. 77–93. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89363-1_5
13. Liu, S., Ölveczky, P., Wang, Q., Meseguer, J.: Formal modeling and analysis of the Walter transactional data store, report available at https://sites.google.com/site/siliunobi/walter
14. Liu, S., Ganhotra, J., Rahman, M., Nguyen, S., Gupta, I., Meseguer, J.: Quantitative analysis of consistency in NoSQL key-value stores. Leibniz Trans. Embed. Syst. **4**(1), 03:1–03:26 (2017)
15. Liu, S., Ölveczky, P.C., Rahman, M.R., Ganhotra, J., Gupta, I., Meseguer, J.: Formal modeling and analysis of RAMP transaction systems. In: SAC 2016. ACM (2016)

16. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 332–347. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11737-9_22
17. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)
18. Ölveczky, P.C.: Formalizing and validating the P-Store replicated data store in Maude. In: James, P., Roggenbach, M. (eds.) WADT 2016. LNCS, vol. 10644, pp. 189–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72044-9_13
19. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_26
20. Sen, K., Viswanathan, M., Agha, G.A.: VESTA: a statistical model-checker and analyzer for probabilistic systems. In: QEST 2005. IEEE Computer Society (2005)
21. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: SOSP 2011. ACM (2011)
22. Younes, H.L.S., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. Inf. Comput. **204**(9), 1368–1409 (2006)
23. Zhang, I., Sharma, N.K., Szekeres, A., Krishnamurthy, A., Ports, D.R.K.: Building consistent transactions with inconsistent replication. In: SOSP 2015. ACM (2015)

Extending Timbuk to Verify Functional Programs

Thomas Genet(✉), Tristan Gillard, Timothée Haudebourg, and Sébastien Lê Cong

Univ Rennes/Inria/CNRS/IRISA, Campus Beaulieu, 35042 Rennes Cedex, France
Thomas.genet@irisa.fr

Abstract. Timbuk implements the Tree Automata Completion algorithm whose purpose is to over-approximate sets of terms reachable by a term rewriting system. Completion is parameterized by a set of equations defining which terms are equated in the approximation. In this paper we present two extensions of Timbuk which permit us to automatically verify safety properties on functional programs. The first extension is a language, based on regular tree expressions, which eases the specification of the property to prove on the program. The second extension automatically generates a set of equations adapted to the property to prove on the program.

1 Motivations

Term Rewriting Systems (TRS for short) are a well known model of functional programs. This model has been used for different kind of analysis ranging from model-checking [4], to static analysis [16] and from termination analysis [13] to complexity analysis [1]. In this paper we focus on static analysis of safety properties on functional programs. Let us illustrate this on a simple example. Assume that we want to analyze the following `delete` OCaml function:

```
let rec delete x l= match l with
  | [] -> []
  | h::t -> if h=x then (delete x t) else h::(delete x t);;
```

In Timbuk [9], this program will be translated in the following TRS, where `ite` encodes the if-then-else construction and `eq` encodes a simple equality on two arbitrary constant symbols `A` and `B`. The **Ops** section defines the symbols with their arity, the **Const** section defines the *constructor* symbols (symbols that are not associated with a function), the **Vars** section defines variables and the **TRS** section associates the name of the TRS with its rules. In the following, we denote by $\mathcal{F}$ the set of symbols defined in the **Ops** section and $\mathcal{T}(\mathcal{F})$ the set of ground terms built on $\mathcal{F}$. We denote by $\mathcal{C}$ the set of constructor symbols defined by **Const**, and $\mathcal{T}(\mathcal{C})$ the set of ground terms defined on $\mathcal{C}$.

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 153–163, 2018.
https://doi.org/10.1007/978-3-319-99840-4_9

```
Ops delete:2 cons:2 nil:0 A:0 B:0 ite:3 true:0 false:0 Eq:2
Const A B nil cons true false
Vars X Y Z
TRS R
  delete(X,nil)->nil
  delete(X,cons(Y,Z))->ite(eq(X,Y),delete(X,Z),cons(Y,delete(X,Z)))
  ite(true,X,Y)->X
  ite(false,X,Y)->Y
  eq(A,A)->true   eq(A,B)->false   eq(B,A)->false   eq(B,B)->true
```

Let us denote by L the set of all possible lists of `A`'s and `B`'s. On this program, we are interested in proving that for all `l` $\in L$, `delete(A,l)` can only result into a list where `A` does not occur. This is equivalent to proving that for all `l` $\in L$, `delete(A,l)` never rewrites to a list containing an `A`. This can be done using reachability analysis on rewriting with the above TRS $\mathcal{R}$. We denote by I the set of all *initial terms*, i.e., $I = \{ \texttt{delete(A,l)} \mid \texttt{l} \in L\}$ and let Bad be the set of lists containing at least one `A`. We denote by $\mathcal{R}^*(I)$ the set of terms reachable by rewriting terms of I with $\mathcal{R}$, i.e., $\mathcal{R}^*(I) = \{t \mid s \in I \text{ and } s \rightarrow_{\mathcal{R}}{}^* t\}$, where $\rightarrow_{\mathcal{R}}{}^*$ is the reflexive and transitive closure of $\rightarrow_{\mathcal{R}}$. If $\mathcal{R}^*(I) \cap Bad = \emptyset$ then there is no way to rewrite a term of the form `delete(A,l)` with `l` $\in L$ into a list containing an `A` and the property is also true on the functional program. Note that the property proved on the TRS is stronger than the property proved on the functional program. In particular, it is independent of the evaluation strategy: it can be call-by-value as well as call-by-name. Thus, the property is true for OCaml as well as for Haskell programs.[1] This paper presents two extensions of Timbuk making the above analysis possible and automatic.

- The first extension are *simplified regular tree expressions* which let the user easily and intuitively define the set of initial terms I.
- The second extension automatically generates abstraction equations, using algorithms described in [8,10]. This makes it possible to automatically build a regular over-approximation App of $\mathcal{R}^*(I)$ such that $App \cap Bad = \emptyset$, if it exists.

In Sect. 2, we define simplified regular expressions. In Sect. 3, we explain why abstraction equations are necessary and we show how to generate them in Sect. 4. In Sect. 5, we show how to interact with Timbuk in order to carry out a complete analysis, as the one shown above. Finally, in Sect. 6, we conclude and give further research directions.

2 Simplified Regular Tree Expressions

We defined the TRS but we still need to define the set of initial terms I in Timbuk. Until now, it could only be defined using a tree automaton [5]. Defining I with this formalism is possible but it is error-prone and lacks readability.

[1] When the analysis depends on the evaluation strategy, completion can be extended to take it into account [12].

As in the case of word languages, there exists an alternative representation for regular tree languages: *regular tree expressions* [5]. However, unlike classical regular expressions for words, regular tree expressions are difficult to read and to write. For instance, the regular tree expression defining terms of the form $f(g^n(a), h^m(b))$ with $n, m \in \mathbb{N}$ is $f(g(\Box_1)^{*,\Box_1}.{}_{\Box_1}a, h(\Box_2)^{*,\Box_2}.{}_{\Box_2}b)$, where $\Box_1$ and $\Box_2$ are new constants. In this expression, the sub-expression $g(\Box_1)^{*,\Box_1}.{}_{\Box_1}a$ represents the language $g^n(a)$. The effect of ${}^{*,\Box_1}$ is to iteratively replace $\Box_1$ by $g(\Box_1)$, and the effect of $.{}_{\Box_1}a$ is to replace $\Box_1$ by a. Regular tree expressions are expressive enough to define any regular tree language. To be complete w.r.t. regular tree languages, this formalism needs named placeholders (like $\Box_1$ and $\Box_2$ above) because the effect of the iteration symbol $*$ depends on the position where it occurs. However, named placeholders make regular tree expressions difficult to read and to write, even if they define simple languages. For instance, the set $I = \{$`delete(A,l)` $\mid$ `l` $\in L\}$ defined above can be written $delete(A, cons((A|B), \Box_1)^{*,\Box_1}.{}_{\Box_1}nil)$ where $\Box_1$ is a new constant.

In this paper, we propose a new formalism for defining regular tree languages: *simplified regular tree expression* (SRegexp for short). Those expressions are not complete w.r.t. regular languages but are easier to read and to write. For instance, the set I is defined by the SRegexp `delete(A,[cons((A|B),*|nil)])`. Those regular expressions are defined using only 3 operators: '`|`' to build the union of two languages, '`*`' to iterate a pattern and the optional brackets '`[ ... ]`' to define the scope of the embedded `*`. The SRegexp `cons((A|B),*|nil)` repeats the pattern `cons(A, _)` or `cons(B,_)` as long as possible and terminates by `nil`. Thus, it defines the language {`nil, cons(A,nil), cons(B,nil), cons(A,cons(A,nil)), cons(A,cons(B,nil)),` ...}. The brackets define the scope of the pattern to repeat with `*`. In the SRegexp `delete(A,[cons((A|B),*|nil)])`, the iteration applies on `cons(A, _)` or `cons(B,_)` but not on `delete(A,_)`. Thus, this expression represents the language {`delete(A, nil), delete(A,cons(A,nil)), delete(A,cons(B,nil)), ...`}.[2]

We implemented SRegexp in Timbuk together with a translation to standard regular tree expressions. We also implemented the translation from regular tree expressions to tree automata defined in [18]. Thus, from a SRegexp I, Timbuk can automatically generate a tree automaton $\mathcal{A}$ whose recognized language $\mathcal{L}(\mathcal{A})$ is equal to I. We also implemented the converse operations: tree automata to regular expression using the algorithm [15] and regular tree expressions to SRegexp. Note that, since SRegexp are incomplete w.r.t. regular tree languages, conversion from regular tree expression to SRegexp may fail. Thus, the over-approximation of reachable terms computed by Timbuk is presented as a SRegexp if it is possible, or as a tree automaton otherwise.

[2] See the page http://people.irisa.fr/Thomas.Genet/timbuk/funExperiments/simplifiedRegexp.html for more examples.

3 The Need for Abstraction Equations

Starting from $\mathcal{R}$ and $I = \mathcal{L}(\mathcal{A})$, computing $\mathcal{R}^*(I)$ is not possible in general [14]. Nevertheless, if $\mathcal{R}$ is a left-linear TRS then $\mathcal{R}^*(I)$ can be over-approximated with tree automata completion [6]. From $\mathcal{A}$ and $\mathcal{R}$, completion builds a tree automaton $\mathcal{A}^*$ such that $\mathcal{L}(\mathcal{A}^*) \supseteq \mathcal{R}^*(I)$. If Bad is regular, to prove $\mathcal{R}^*(I) \cap Bad = \emptyset$, it is enough to check that $\mathcal{L}(\mathcal{A}^*) \cap Bad = \emptyset$, which can be done efficiently [5]. For this technique to succeed, the precision of the approximation $\mathcal{A}^*$ is crucial. For instance, $\mathcal{L}(\mathcal{A}^*) = \mathcal{T}(\mathcal{F})$ is a valid regular over-approximation but it cannot be used to prove any safety property since it also contains Bad. In Timbuk, approximations are defined using sets of abstraction equations, following [11,20].

Example 1. Let L be the set of terms defined with the symbol s of arity 1 and the constant symbol 0. Let X be a variable. The effect of the equation $s(s(X)) = s(X)$ is to merge in the same equivalence class terms $s(s(0))$ and $s(0)$, $s(s(s(0)))$ and $s(s(0))$, etc. Thus, with this single equation, $L/_{=_E}$ consists of only two equivalence classes: a class containing only 0 and the class containing all the other natural numbers $\{s(0), s(s(0)), \ldots\}$. An equation $s(X) = X$ would define a single equivalence class containing all natural numbers. It would thus define a rougher abstraction. An equation $s(s(X)) = X$ defines two equivalence classes: the class of even numbers $\{0, s(s(0)), \ldots\}$ and the class of odd numbers $\{s(0), s(s(s(0))), \ldots\}$.

For completion to terminate, the set $\mathcal{T}(\mathcal{F})/_{=_E}$ (E-equivalence classes of $\mathcal{T}(\mathcal{F})$) has to be finite [8]. When dealing with functional programs, this restriction can be relaxed as follows. Functional programs manipulate sorted terms and the associated TRSs preserve sorts. Provided that equations also preserve sorts, having a finite set $\mathcal{T}(\mathcal{F})^S/_{=_E}$, where $\mathcal{T}(\mathcal{F})^S$ is the set of well-sorted terms, is enough. Besides, since well-sorted terms define a regular language, this information can be provided to Timbuk using tree automata, regular expressions or SRegexp.

Example 2. Let us consider the set L of well-sorted lists of A and B. The set L is the regular language associated with the SRegexp $cons((A|B), *|nil)$. Let X, Y, Z be variables. The set $E = \{cons(X, cons(Y, Z)) = cons(Y, Z)\}$ defines a set of E-equivalence classes $L/_{=_E}$ with three classes: one class only contains nil, one class contains all lists ending with an A and the last class contains all lists ending with a B.

Going back to the `delete` example that we want to analyze, with set $E = \{cons(X, cons(Y, Z)) = cons(Y, Z)\}$, $L/_{=_E}$ is finite but $\mathcal{T}(\mathcal{F})^S/_{=_E}$ may not be. For instance, terms $delete(A, nil)$, $delete(A, delete(A, nil))$, etc. are all in separate equivalence classes. Again, we can take advantage from the fact that `delete` is a functional program and relax the termination condition of completion by focusing it on the data manipulated by the program. Instead of asking for finiteness of $\mathcal{T}(\mathcal{F})^S/_{=_E}$, we only require finiteness of $\mathcal{T}(\mathcal{C})^S/_{=_E}$, where $\mathcal{T}(\mathcal{C})^S$ is the set of *well-sorted constructor terms*. Let us note E_c

the above set of equations $\{cons(X, cons(Y, Z)) = cons(Y, Z)\}$. As shown in Example 2, E_c defines a finite set of equivalence classes on $\mathcal{T}(\mathcal{C})^S$, i.e., lists of A's and B's.[3] Provided that `delete` is a terminating and complete functional program, it is possible to extend E_c so that completion terminates. This has been shown for first-order functional programs [7] and for higher-order functional programs [10]. The extension of E_c consists in adding two sets of equations $E_{\mathcal{R}} = \{l = r \mid l \rightarrow r \in \mathcal{R}\}$ and $E_r = \{f(X_1, \ldots, X_n) = f(X_1, \ldots, X_n) \mid f \in \mathcal{F}$, arity of f is n, and $X_1, \ldots, X_n$ are variables$\}$. Since $E_{\mathcal{R}}$ and E_r are fixed by the program, the precision of the approximation only depends on the equivalence classes defined by E_c. Thus, to explore approximations, it is enough to explore all possible E_c.

4 Generating Abstraction Equations E_c

Additionally to the fact that (1) $\mathcal{T}(\mathcal{C})^S/_{=E_c}$ has to be finite, the termination theorems of [7,10] imposes additional constraints on E_c. Equations in E_c have to be *contracting*, i.e., they are of the form $u = u|_p$ where (2) $u|_p$ is a strict subterm of u and (3) $u|_p$ has the same sort as u.[4] Conditions (2) and (3) makes it possible to prune the search space of equations in E_c. For instance the following equations do not need to be considered: $cons(X, Y) = Z$ because of condition (2), $cons(X, cons(Y, Z)) = cons(X, Z)$ because of condition (2), $cons(X, Y) = X$ because of condition (3).

Timbuk implements two different algorithms to explore the space of possible E_c. Those algorithms are parameterized by a natural number $k \in \mathbb{N}$ and, for a given k, they generate a set $EC(k)$ of possible E_c. By increasing k, we increase the precision of equations sets E_c in $EC(k)$. The first algorithm is based on covering sets [17] and generates contracting equations with variables [10]. In this algorithm k defines the depth of the covering set used to generate the equations. From a covering set S, we generate all equations sets $E_c = \{u = u|_p \mid u \in S\}$ satisfying conditions (1) to (3).

Example 3. Let X be a variable and $\mathcal{T}(\mathcal{C})^S$ be the set of well-sorted constructor terms defined with symbol s of arity 1 and the constant symbol 0. For $k = 1$, the covering set is $\{s(X), 0\}$ and $EC(1) = \{\{s(X) = X\}\}$. For $k = 2$, the covering set is $\{s(s(X)), s(0), 0\}$ and $EC(2) = \{\{s(s(X)) = X\}, \{s(s(X)) = s(X)\}, \{s(0) = 0\}, \{s(0) = 0, s(s(X)) = X\}, \{s(0) = 0, s(s(X)) = s(X)\}\}$.

The second algorithm generates *ground* contracting equations [8]. In this algorithm k represents the number of equivalence classes expected in $\mathcal{T}(\mathcal{C})^S/_{=E_c}$.

[3] In fact, in $\mathcal{T}(\mathcal{C})^S$ there are also terms `true` and `false` but they cannot be embedded in lists. Thus, each of them defines its own equivalence class. In the end, in $\mathcal{T}(\mathcal{C})^S/_{=E_c}$ there are 5 equivalence classes.

[4] Note that the sort information can be inferred from the tree automaton recognizing well-sorted terms. For instance, the automaton associated to the SRegexp of Example 2 recognizes A and B by into the same state, thus A and B will have the same sort (see automaton `TC` in Sect. 5).

Since equation sets have to be ground and meet conditions (2) and (3), we can finitely enumerate all the possible equations sets E_c for a given k.

Example 4. Let $\mathcal{T}(\mathcal{C})^S$ be the set of well-sorted constructor terms defined with symbol s of arity 1 and the constant symbol 0. For $k = 1$ the set $EC(1) = \{\{s(0) = 0\}\}$. For $k = 2$, the set $EC(2) = \{\{s(s(0)) = 0\}, \{s(s(0)) = s(0)\}$.

A systematic way to build ground $EC(k)$, based on tree automata enumeration, is given in [8]. Using the first or second algorithm to generate $EC(k)$, to prove that there exists a tree automaton $\mathcal{A}^*$ over-approximating $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ and such that $\mathcal{L}(\mathcal{A}^*) \cap Bad = \emptyset$, we run the following algorithm:

1. Start with $k = 1$
2. Build $EC(k)$
3. Pick one E_c in $EC(k)$
4. Complete $\mathcal{A}$ into $\mathcal{A}^*$ using $\mathcal{R}$ and $E_c \cup E_{\mathcal{R}} \cup E_r$
5. If $\mathcal{L}(\mathcal{A}^*) \cap Bad = \emptyset$ then verification is successful Otherwise, if $EC(k)$ not empty, pick a new E_c in $EC(k)$ and go to 4.
6. When $EC(k)$ is empty, increment k and go to 2.

It has been shown in [8] that the ground enumeration of $EC(k)$ is complete w.r.t. tree automata that are closed by $\mathcal{R}$-rewriting. Thus, if there exists such a $\mathcal{A}^*$, the above iterative algorithm will find it. However, on properties that cannot be shown using a regular approximation, such as [2], this algorithm may diverge.

5 Interacting with Timbuk

Download http://people.irisa.fr/Thomas.Genet/timbuk/timbuk3.2.tar.gz and compile and install Timbuk 3.2. The online version of Timbuk does not integrate all the features presented here. In Timbuk's archive, the full specification of the `delete` example can be found in the file `FunExperiments/deleteBasic.txt`.

```
Ops delete:2 cons:2 nil:0 A:0 B:0 ite:3 true:0 false:0 Eq:2
Const A B nil cons true false
Vars X Y Z
TRS R
   delete(X,nil)->nil
   delete(X,cons(Y,Z))->ite(eq(X,Y),delete(X,Z),cons(Y,delete(X,Z)))
   ite(true,X,Y)->X
   ite(false,X,Y)->Y
   eq(A,A)->true   eq(A,B)->false   eq(B,A)->false   eq(B,B)->true
SRegexp A0
   delete(A,[cons((A|B),*|nil)])
Automaton TC
  States qe ql qb
  Final States qe ql qb
  Transitions
    A->qe  B->qe  nil->ql  cons(qe,ql)->ql  true->qb  false->qb
Patterns
   cons(A,_)
```

This file contains the TRS, the SRegexp presented above and a tree automaton named `TC` which defines well-sorted constructor terms as explained in Example 2. This automaton is used to prune equation generation. Note that this automaton could be inferred from the typing information of the functional program. Here, the automaton `TC` states that lists are built with `cons` and `nil`, that elements of the list are either `A` or `B`, and that `true` and `false` are of the same type but cannot appear in a list. Thus, ill-typed terms of the form $cons(nil, true)$ are not considered for equation generation. Finally, the `Patterns` section defines the set Bad of terms that should not be reachable. Currently, the pattern section is limited to terms or patterns (terms with holes '_') and cannot handle SRegexp or automata. In the present example, we only consider a subset of bad terms: terms of the form `cons(A,_)`, i.e., lists starting by `A`. Assuming that your working directory is `FunExperiments`, you can run Timbuk on this example by typing: `timbuk --fung 30 deleteBasic.txt`. Where `--fung` is the option triggering ground equation generation (the second algorithm for generating $EC(k)$) and `30` is a maximal number of completion steps. We get the following output:

```
Generated equations:
--------------------
cons(A,cons(A,nil)) = cons(A,nil)
cons(B,cons(A,nil)) = cons(A,nil)
cons(B,nil) = nil
B = B
nil = nil
delete(X,Y) = delete(X,Y)
A = A
true = true
cons(X,Y) = cons(X,Y)
false = false
ite(X,Y,Z) = ite(X,Y,Z)
eq(X,Y) = eq(X,Y)
eq(A,A) = true
eq(A,B) = false
eq(B,A) = false
eq(B,B) = true
delete(X,nil) = nil
delete(X,cons(Y,Z)) =
  ite(eq(X,Y),delete(X,Z),cons(Y,delete(X,Z)))
ite(true,X,Y) = X
ite(false,X,Y) = Y

Regular expression:
-------------------
[cons(B, *|nil)]

Proof done!
-----------
Completion time: 0.006595 seconds
```

The three first generated equations belong to E_c, reflexive equations of the form `B = B`, `nil = nil`, ... belong to E_r and the last eight equations belong to $E_{\mathcal{R}}$. The set $\mathcal{T}(\mathcal{C})^S/_{=E_c}$ has two equivalence classes: the class containing nil and all lists containing only B's and the class of lists containing at least one A. Thus, the effect of E_c is to forget any B and preserve any A that appears in a list. Using the `--fun` option instead of `--fung` while running Timbuk, triggers the first algorithm for generating $EC(k)$, i.e., E_c with variables. On this example, the generated E_c part has two equations instead of three: `cons(X,cons(A,Y)) = cons(A,Y)` and `cons(B,X) = X`. The effect of this set E_c is the same as the ground E_c above. Indeed, this E_c splits lists into two equivalence classes: the class of lists without A's and the class of lists with at least one A.

Finally, in Timbuk's output, `Proof done!` means that Timbuk manages to build a regular approximation of $\mathcal{R}^*(I)$ that contains no term of the `Patterns` section. Timbuk outputs the resulting simplified regular expression `[cons(B,`

`*|nil)]`. This proves that results are lists without any occurrence of `A`'s. Here, one can read the outputted SRegexp to check that the property is true. However, this can be difficult when the outputted SRegexp is more complex. Thus, on most examples, we use additional predicates to check properties like it is commonly done with proof assistants. On our previous example, given a predicate `member` (testing membership on lists), we can check that terms of the language `member(A,delete(A,cons((A|B),*|nil)))` never rewrite to `true`. We can also check the dual property expected on `delete`: deleting A's should not delete all B's. We hope to check this property using initial terms `member(B,delete(A, [cons((A|B),*|nil)]))` and a `patterns` section set to `false`. However, the property is not true and, during completion, Timbuk finds a counterexample:

```
Found a counterexample:
-----------------------
Term member(B,delete(A,nil)) rewrites to a forbidden pattern
```

For the property to hold, lists in initial terms should contain at least one B:

`member(B,delete(A,[cons((A|B),*|[cons(B,*|[cons((A|B),*|nil)])])]))`

Using this initial set of terms, Timbuk succeeds to do the proof and produces a slightly different E_c: `cons(A,cons(B,nil)) = cons(B,nil)`, `cons(B,cons(B,nil)) = cons(B,nil)`, `cons(A,nil) = nil`. This time, E_c forgets about A's and preserves B's. More than 20 other examples (with ground/non-ground equations generation) can be found on the Timbuk page http://people.irisa.fr/Thomas.Genet/timbuk/funExperiments/, including functions on lists, trees, sorting functions, higher-order functions, etc.

6 Conclusion and Further Research

We know that completion is terminating on higher-order functional programs thanks to the recent result of [10]. Besides, we also know that ground equation generation of E_c is complete w.r.t. tree automata that are closed by $\mathcal{R}$ [8]. In other words, if there exists a tree automaton $\mathcal{A}^*$, closed by $\mathcal{R}$ and over-approximating the set of reachable terms, then it will eventually be found by generating ground equations. With the first algorithm where equations of E_c may contain variables, we do not have a similar completeness result, yet. However, generating equations with variables remains an interesting option because the set E_c can be smaller. This is the case in the previous example where E_c with variables defines the same set of equivalence classes but with fewer equations.

From a theoretical perspective, Tree Automata Completion can be seen as an alternative to well-established higher-order model-checking techniques like PMRS [21] or HORS [19] to verify higher-order functional programs. Timbuk implements Tree Automata Completion but was missing several features for those theoretical results to be usable in practice. First, stating the property to prove using a tree automaton was error-prone and lacked readability. Using simplified regular expressions significantly improves this step and makes property

definition closed to what is generally used in a proof assistant. Second, equations which are necessary to define the approximation, had to be given by the user [7]. Now, Timbuk can automatically generate a set of equations adapted to a given verification objective. Combining those two extensions makes Timbuk a competitive alternative to higher-order model checking tools like [21] and [19].

In those model-checking tools and in Timbuk, the properties under concern are "regular properties", i.e. properties proven on regular languages. Those regular properties are stronger than what offers tests (they prove a property on an infinite set of values) but weaker than what can be proven using induction in a proof assistant. However, unlike proof assistants, Timbuk does not require to write lemmas or proof scripts to prove a regular property. An interesting research direction is to explore how to lift those regular properties to general properties. In other words, how to build a proof that ∀x l. `not(member(x,delete(x,l)))` from the fact that all terms from `member(A,delete(A,cons((A|B),*|nil)))` rewrite to `false`. We believe that this is possible by taking advantage of parametricity such as in [22]. This is ongoing work.

In this paper, the verification is performed on a TRS representing the functional program. To directly perform the verification on real functional programs rather than on TRSs, we need a transformation. We could reuse the HOCA transformation of [1]. However, it does not take the priorities of the pattern matching rules of the functional program into account when producing the TRS. Furthermore, this translation needs to be certified, i.e., we need a formal proof that the behavior of the outputted TRS $\mathcal{R}$ covers all the possible behaviors of the functional program. With such a proof on $\mathcal{R}$, if Timbuk can prove that no term of `member(A,delete(A,cons((A|B),*|nil)))` can be rewritten to `true` with $\mathcal{R}$, then we have a similar property on the functional program.

The equation generation process does not cover all TRSs but only TRSs encoding terminating, complete, higher-order, functional programs. We currently investigate how to generate equations without the termination and completeness restrictions on the program. Another research direction is to extend this verification principle to more general theorems. For the moment, theorems that can be proved using Timbuk need to have a regular model. For instance, Timbuk is able to prove the theorem `member(A,delete(A,l))` $\not\to_{\mathcal{R}}^*$ `true` for all lists `l=cons((A|B),*|nil)` because the language of terms reachable from the initial language `member(A,delete(A,cons((A|B),*|nil)))` is, itself, regular. Assume that we have a predicate `eq` encoding equality on lists. To prove a theorem of the form `eq(delete(A,l),l)` $\not\to_{\mathcal{R}}^*$ `false` for all list `l=cons(B,*|nil)`, the language of reachable terms is no longer regular. However, recent advances in completion-based techniques for non-regular languages [3] should make such verification goals reachable.

Acknowledgments. Many thanks to the anonymous referees for their valuable comments.

References

1. Avanzini, M., Dal Lago, U., Moser, G.: Analysing the complexity of functional programs: higher-order meets first-order. In: ICFP 2015, pp. 152–164. ACM (2015)
2. Boichut, Y., Héam, P.-C.: A theoretical limit for safety verification techniques with regular fix-point computations. IPL **108**(1), 1–2 (2008)
3. Boichut, Y., Chabin, J., Réty, P.: Towards more precise rewriting approximations. In: Dediu, A.-H., Formenti, E., Martín-Vide, C., Truthe, B. (eds.) LATA 2015. LNCS, vol. 8977, pp. 652–663. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15579-1_51
4. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
5. Comon, H., et al.: Tree Automata Techniques and Applications (2008). http://tata.gforge.inria.fr
6. Genet, T.: Decidable approximations of sets of descendants and sets of normal forms. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 151–165. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0052368
7. Genet, T.: Termination criteria for tree automata completion. J. Log. Algebraic Methods Program. **85**(1), 3–33 (2016). Part 1
8. Genet, T.: Automata Completion and Regularity Preservation. Technical report, INRIA (2017). https://hal.inria.fr/hal-01501744
9. Genet, T., Boichut, Y., Boyer, B., Gillard, T., Haudebourg, T., Lê Cong, S.: Timbuk 3.2 - a Tree Automata Library. IRISA/Université de Rennes 1 (2017). http://people.irisa.fr/Thomas.Genet/timbuk/
10. Genet, T., Haudebourg, T., Jensen, T.: Verifying higher-order functions with tree automata. In: Baier, C., Dal Lago, U. (eds.) FoSSaCS 2018. LNCS, vol. 10803, pp. 565–582. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89366-2_31
11. Genet, T., Rusu, R.: Equational tree automata completion. J. Symb. Comput. **45**, 574–597 (2010)
12. Genet, T., Salmon, Y.: Reachability analysis of innermost rewriting. In: RTA 2015, volume 36 of LIPIcs, Warshaw. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
13. Giesl, J.: Termination analysis for functional programs using term orderings. In: Mycroft, A. (ed.) SAS 1995. LNCS, vol. 983, pp. 154–171. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60360-3_38
14. Gilleron, R., Tison, S.: Regular tree languages and rewrite systems. Fundam. Inform. **24**, 157–175 (1995)
15. Guellouma, Y., Mignot, L., Cherroun, H., Ziadi, D.: Construction of rational expression from tree automata using a generalization of Arden's lemma. CoRR, abs/1501.07686 (2015)
16. Jones, N.D., Andersen, N.: Flow analysis of lazy higher-order functional programs. Theor. Comput. Sci. **375**(1–3), 120–136 (2007)
17. Kounalis, E.: Testing for the ground (co-)reducibility property in term-rewriting systems. TCS **106**(1), 87–117 (1992)
18. Kuske, D., Meinecke, I.: Construction of tree automata from regular expressions. RAIRO Theor. Inform. Appl. **45**(3), 347–370 (2011)
19. Matsumoto, Y., Kobayashi, N., Unno, H.: Automata-Based abstraction for automated verification of higher-order tree-processing programs. In: Feng, X., Park, S. (eds.) APLAS 2015. LNCS, vol. 9458, pp. 295–312. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26529-2_16

20. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. TCS **403**(2–3), 239–264 (2008)
21. Ong, L., Ramsay, S.: Verifying higher-order functional programs with pattern-matching algebraic data types. In: POPL 2011. ACM (2011)
22. Wadler, P.: Theorems for free! In: Proceedings of FPCA 1989, pp. 347–359. ACM (1989)

Generalized Rewrite Theories and Coherence Completion

José Meseguer(✉)

Department of Computer Science,
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
meseguer@illinois.edu

Abstract. A new notion of generalized rewrite theory suitable for symbolic reasoning and generalizing the standard notion in [3] is motivated and defined. Also, new requirements for *symbolic executability* of generalized rewrite theories that extend those in [8] for standard rewrite theories, including a generalized notion of *coherence*, are given. Finally, symbolic executability, including coherence, is both ensured and made available for a wide class of such theories by automatable *theory transformations.*

Keywords: Rewriting · Coherence · Variants · Symbolic execution

1 Introduction

Symbolic methods are used to reason about concurrent systems specified by rewrite theories in many ways, including: (i) cryptographic protocol verification, e.g., [10], (ii) logical LTL model checking, e.g., [2], (iii) rewriting modulo SMT and related approaches, e.g., [1,23], (iv) inductive theorem proving and program verification, e.g., [12,16], and (v) reachability logic theorem proving, e.g., [17,25, 26]. One key issue is that the rewrite theories used in several of these approaches go *beyond* the standard notion of rewrite theory in, say [3], and also beyond the *executability requirements* in, say, [8]. For example: (1) conditions in rules are not just conjunctions of equations, but quantifier-free (QF) formulas in an, often decidable, *background theory* T (e.g., Presburger arithmetic); and (2) the rewrite rules may model *open systems* interacting with an environment, so that they may have extra variables in their righthand sides [23]. Furthermore, each of the approaches just mentioned uses *different* assumptions about the rewrite theories they handle: no general notion has yet been proposed.

There are also unsolved issues about *symbolic executability*: even though symbolic execution methods in some ways *relax* executability requirements, in other ways they impose *strong restrictions* on the rewrite rules to be executed. For example, in narrowing-based reachability analysis the presence of extra variables in righthand sides of rules is unproblematic. Nevertheless, unless *both* the lefthand and righthand sides of a rewrite rule are terms in an equational theory having a *finitary* unification algorithm, symbolic reachability analysis becomes

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 164–183, 2018.
https://doi.org/10.1007/978-3-319-99840-4_10

extremely difficult and is usually outside the scope of current methods. There is also plenty of *terra incognita*. For example, we all optimistically assume and require that the rewrite theories we are going to symbolically execute are of course *coherent* [8,28]. But no theory of coherence, or methods for guaranteeing it, have yet been developed for these new kinds of theories.

The upshot of all this is that, as usual, the new wine of symbolic reasoning requires new wineskins. This work is all about such new wineskins. It asks, and provides answers for, two main questions: (1) How can the notion of rewrite theory be *generalized* to support symbolic reasoning? and (2) What are the appropriate *symbolic executability requirements* needed for such rewrite theories; and how can they be *ensured* for, and made available to, a widest possible class of theories?

Outline and Main Contributions. Section 2 gathers preliminaries. Section 3 motivates and presents a notion of *generalized rewrite theory* suitable for symbolic reasoning and subsuming the standard notion as a special case. It also defines an *initial model semantics* for such theories in an associated category of algebraic transition systems. Finally, it uses such a semantics to identify *symbolic executability requirements*, including a generalized notion of *coherence* and an easier to check characterization of it. Section 4 then addresses and provides solutions for two related problems: (i) how can (ground) coherence be ensured *automatically* under reasonable requirements? and (ii) how can the class of generalized rewrite theories that can be symbolically executed be made *as wide as possible* by means of adequate *theory transformations*? Note that the answer to question (i) is new even for standard rewrite theories and can be quite useful to semi-automate equational abstractions [22]. This automation method is an interesting instance of what might be called *theoretical dogfooding*, where the new symbolic methods of variant computation [11,21,24] are applied to *complete* a rewrite theory into a ground coherent one. The answer to question (ii) is very general: under mild conditions symbolic executability can be ensured for a wide class of generalized theories by two theory transformations. Related work and conclusions are discussed in Sect. 5. Proofs and full details about examples can be found in the Technical Report [20].

2 Preliminaries on Order-Sorted Algebra and Variants

I present needed preliminaries on order-sorted algebra, logic, and variants. The material is adapted from [19,21]. The presentation is self-contained: only the notions of many-sorted signature and many-sorted algebra, e.g., [9], are assumed.

Definition 1. *An* order-sorted (OS) signature *is a triple* $\Sigma = (S, \leqslant, \Sigma)$ *with* $(S, \leqslant)$ *a poset and* (S, Σ) *a many-sorted signature.* $\widehat{S} = S/{\equiv_{\leqslant}}$*, the quotient of* S *under the equivalence relation* $\equiv_{\leqslant} = (\leqslant \cup \geqslant)^{+}$*, is called the set of* connected components, or kinds *of* $(S, \leqslant)$*. The order* $\leqslant$ *and equivalence* $\equiv_{\leqslant}$ *are extended to sequences of same length in the usual way, e.g.,* $s'_1 \dots s'_n \leqslant s_1 \dots s_n$ *iff* $s'_i \leqslant s_i$*,* $1 \leqslant i \leqslant n$*.* Σ *is called* sensible *if for any two* $f : w \to s, f : w' \to s' \in \Sigma$*, with* w

and w' of same length, we have $w \equiv_{\leqslant} w' \;\Rightarrow\; s \equiv_{\leqslant} s'$. A many-sorted signature *Σ is the special case where the poset $(S, \leqslant)$ is discrete, i.e., $s \leqslant s'$ iff $s = s'$.*

For connected components $[s_1], \ldots, [s_n], [s] \in \widehat{S}$

$$f_{[s]}^{[s_1]\ldots[s_n]} = \{f : s'_1 \ldots s'_n \to s' \in \Sigma \mid s'_i \in [s_i],\;\; 1 \leqslant i \leqslant n,\; s' \in [s]\}$$

denotes the family of "subsort polymorphic" operators f. We can extend any $\Sigma = (S, \leqslant, \Sigma)$ to its kind completion *$\widehat{\Sigma} = (S \uplus \widehat{S}, \widehat{\leqslant}, \widehat{\Sigma})$ where: (i) $\widehat{\leqslant}$ is the least partial order extending $\leqslant$ such that $s < [s]$ for each $s \in S$, and (ii) we add to each family of subsort polymorphic operators $f_{[s]}^{[s_1]\ldots[s_n]}$ in Σ the operator $f : [s_1] \ldots [s_n] \to [s]$.* □

Definition 2. *For $\Sigma = (S, \leqslant, \Sigma)$ an OS signature, an* order-sorted Σ-algebra *A is a many-sorted (S, Σ)-algebra A such that:*

- *whenever $s \leqslant s'$, then we have $A_s \subseteq A_{s'}$, and*
- *whenever $f : w \to s, f : w' \to s' \in f_{[s]}^{[s_1]\ldots[s_n]}$ and $\overline{a} \in A^w \cap A^{w'}$, then we have $f_A^{w,s}(\overline{a}) = f_A^{w',s'}(\overline{a})$, where $A^{s_1 \ldots s_n} = A_{s_1} \times \ldots \times A_{s_n}$.*

A Σ-homomorphism *$h : A \to B$ is a many-sorted (S, Σ)-homomorphism such that $([s] = [s'] \wedge a \in A_s \cap A_{s'}) \Rightarrow h_s(a) = h_{s'}(a)$. This defines a category* $\mathbf{OSAlg}_{\Sigma}$. **Notation***: $h : A \cong B$ denotes an isomorphism $h : A \to B$.* □

Theorem 1 *[19]. The category $\mathbf{OSAlg}_{\Sigma}$ has an initial algebra. Furthermore, if Σ is sensible, then the term algebra T_{Σ} with:*

- *if $a : \epsilon \to s$ then $a \in T_{\Sigma,s}$ (ϵ denotes the empty string),*
- *if $t \in T_{\Sigma,s}$ and $s \leqslant s'$ then $t \in T_{\Sigma,s'}$,*
- *if $f : s_1 \ldots s_n \to s$ and $t_i \in T_{\Sigma,s_i}$ $1 \leqslant i \leqslant n$, then $f(t_1, \ldots, t_n) \in T_{\Sigma,s}$,*

is initial, i.e., there is a unique Σ-homomorphism to each Σ-algebra.

For $[s] \in \widehat{S}$, $T_{\Sigma,[s]}$ denotes the set $T_{\Sigma,[s]} = \bigcup_{s' \in [s]} T_{\Sigma,s'}$. T_{Σ} will (ambiguously) denote: (i) the term algebra; (ii) its underlying S-sorted set; and (iii) the set $T_{\Sigma} = \bigcup_{s \in S} T_{\Sigma,s}$. An OS signature Σ is said to *have non-empty sorts* iff for each $s \in S$, $T_{\Sigma,s} \neq \emptyset$. An OS signature Σ is called *preregular* [14] iff for each $t \in T_{\Sigma}$ the set $\{s \in S \mid t \in T_{\Sigma,s}\}$ has a least element, denoted $ls(t)$. *We will assume throughout that Σ has non-empty sorts and is preregular.*

An S-sorted set $X = \{X_s\}_{s \in S}$ of *variables*, satisfies $s \neq s' \Rightarrow X_s \cap X_{s'} = \emptyset$, and the variables in X are always assumed disjoint from all constants in Σ. The *Σ-term algebra* on variables X, $T_{\Sigma}(X)$, is the *initial algebra* for the signature $\Sigma(X)$ obtained by adding to Σ the variables X *as extra constants.* Since a $\Sigma(X)$-algebra is just a pair (A, α), with A a Σ-algebra, and α an *interpretation in A of the constants* in X, i.e., an S-sorted function $\alpha \in [X {\to} A]$, the $\Sigma(X)$-initiality of $T_{\Sigma}(X)$ can be expressed as the following theorem:

Theorem 2 *(Freeness Theorem). If Σ is sensible, for each $A \in \mathbf{OSAlg}_{\Sigma}$ and $\alpha \in [X {\to} A]$, there exists a unique Σ-homomorphism, $_\alpha : T_{\Sigma}(X) \to A$ extending α, i.e., such that for each $s \in S$ and $x \in X_s$ we have $x\alpha_s = \alpha_s(x)$.*

In particular, when $A = T_{\Sigma}(Y)$, an interpretation of the constants in X, i.e., an S-sorted function $\sigma \in [X \to T_{\Sigma}(Y)]$ is called a *substitution*, and its unique homomorphic extension $_\sigma : T_{\Sigma}(X) \to T_{\Sigma}(Y)$ is also called a substitution. Define $dom(\sigma) = \{x \in X \mid x \neq x\sigma\}$, and $ran(\sigma) = \bigcup_{x \in dom(\sigma)} vars(x\sigma)$. Given variables Z, the substitution $\sigma|_Z$ agrees with σ on Z and is the identity elsewhere.

The first-order language of *equational Σ-formulas* is defined in the usual way: its atoms are *Σ-equations* $t = t'$, where $t, t' \in T_{\Sigma}(X)_{[s]}$ for some $[s] \in \widehat{S}$ and each X_s is assumed countably infinite. The set $Form(\Sigma)$ of *equational Σ-formulas* is then inductively built from atoms by: conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and universal ($\forall x{:}s$) and existential ($\exists x{:}s$) quantification with sorted variables $x{:}s \in X_s$ for some $s \in S$. $\varphi \in Form(\Sigma)$ is called *quantifier-free* (QF) iff it does not contain any quantifiers. The literal $\neg(t = t')$ is denoted $t \neq t'$. Given a Σ-algebra A, a formula $\varphi \in Form(\Sigma)$, and an assignment $\alpha \in [Y \to A]$, with $Y = fvars(\varphi)$ the free variables of φ, the *satisfaction relation* $A, \alpha \models \varphi$ is defined inductively in the usual way. By definition, $A \models \varphi$ holds iff for each $\alpha \in [Y \to A]$ $A, \alpha \models \varphi$ holds, where $Y = fvars(\varphi)$ are the free variables of φ. We say that φ is *valid* (or *true*) in A iff $A \models \varphi$. For a subsignature $\Omega \subseteq \Sigma$ and $A \in \mathbf{OSAlg}_{\Sigma}$, the *reduct* $A|_{\Omega} \in \mathbf{OSAlg}_{\Omega}$ agrees with A in the interpretation of all sorts and operations in Ω and discards everything in $\Sigma \setminus \Omega$. If $\varphi \in Form(\Omega)$ we have the equivalence $A \models \varphi \;\Leftrightarrow\; A|_{\Omega} \models \varphi$. Given a set of formulas $\Gamma \subseteq Form(\Sigma)$ we say that $A \in \mathbf{OSAlg}_{\Sigma}$ *satisfies* Γ, written $A \models \Gamma$ iff $\forall \varphi \in \Gamma\; A \models \varphi$. An OS *theory* T is a pair $T = (\Sigma, \Gamma)$ with Σ an OS signature and $\Gamma \subseteq Form(\Sigma)$. For $T = (\Sigma, \Gamma)$, $\mathbf{OSAlg}_{(\Sigma,\Gamma)}$ denotes the full subcategory of $\mathbf{OSAlg}_{\Sigma}$ with objects those $A \in \mathbf{OSAlg}_{\Sigma}$ such that $A \models \Gamma$, called the (Σ, Γ)-*algebras*. Given $T = (\Sigma, \Gamma)$ we call $\varphi \in Form(\Omega)$ a *logical consequence* of T, or *true* in T, denoted $T \models \varphi$ or $\Gamma \models \varphi$, iff $\forall A \in \mathbf{OSAlg}_{(\Sigma,\Gamma)}\; A \models \varphi$. Note that the notion of satisfaction and the Freeness theorem yield the implication $T \models \varphi \;\Rightarrow\; T \models \varphi\theta$ for any substitution θ. Note also that any Σ-algebra A has an associated theory $th(A) = (\Sigma, \{\varphi \in Form(\Sigma) \mid A \models \varphi\})$. A *theory inclusion* $T = (\Sigma, \Gamma) \subseteq (\Sigma', \Gamma') = T'$ holds iff $\Sigma \subseteq \Sigma'$ and $\Gamma' \models \Gamma$, and is called a *conservative extension* iff $\forall \varphi \in Form(\Sigma)\;\; T \models \varphi \;\Leftrightarrow\; T' \models \varphi$. Call $T = (\Sigma, \Gamma)$ and $T' = (\Sigma, \Gamma')$ *semantically equivalent* (denoted $T \equiv T'$) iff $T \subseteq T'$ and $T' \subseteq T$.

An OS *equational theory* (resp. *conditional equational theory*) is an OS theory $T = (\Sigma, E)$ with E a set of Σ-equations (resp. conditional Σ-equations of the form $\bigwedge_{i=1 \ldots n} u_i = v_i \;\Rightarrow\; t = t'$). $\mathbf{OSAlg}_{(\Sigma,E)}$ always has an *initial algebra* $T_{\Sigma/E}$, and *free algebras* $T_{\Sigma/E}(X)$ [19]. The inference system in [19] is *sound and complete* for OS equational deduction, i.e., for any OS equational theory (Σ, E), and Σ-equation $u = v$ we have an equivalence $E \vdash u = v \quad\Leftrightarrow\quad E \models u = v$. Deducibility $E \vdash u = v$ is abbreviated as $u =_E v$, called *E-equality*.

Given a set of equations B used for deduction modulo B, a preregular OS signature Σ is called *B-preregular*[1] iff for each $u = v \in B$ and substitution

[1] If $B = B_0 \uplus U$, with B_0 associativity and/or commutativity axioms, and U identity axioms, the B-preregularity notion can be *broadened* by requiring only that: (i) Σ is B_0-preregular in the standard sense, so that $ls(u\rho) = ls(v\rho)$ for all $u = v \in B_0$

ρ, $ls(u\rho) = ls(v\rho)$. Recall the notation for term positions, subterms, and term replacement from [6]: (i) positions in a term viewed as a tree are marked by strings $p \in \mathbb{N}^*$ specifying a path from the root, (ii) $t|_p$ denotes the subterm of term t at position p, and (iii) $t[u]_p$ denotes the result of *replacing* subterm $t|_p$ at position p by u. Recall also from [18,21] that given an equational theory $(\Sigma, E \uplus B)$ with Σ is *B-preregular*, $=_B$ *decidable*, and such that:

1. each equation $u = v \in B$ is *regular*, i.e., $vars(u) = vars(v)$, and *linear*, i.e., there are no repeated variables in u, and no repeated variables in v;
2. the equations E, when oriented as rewrite rules $\vec{E} = \{t \to t' \mid (t = t') \in E\}$, are *convergent* modulo B, that is, sort-decreasing, strictly B-coherent, confluent, and terminating as rewrite rules modulo B [18],

then we call the rewrite theory $\mathcal{R} = (\Sigma, B, \vec{E})$ (in the sense of [3]) a *decomposition* of the given equational theory $(\Sigma, E \uplus B)$. Given such a decomposition $\mathcal{R} = (\Sigma, B, \vec{E})$, the equality relation $=_{E \uplus B}$ becomes then *decidable* thanks to the rewrite relation $\to_{\vec{E},B}$, where $u \to_{\vec{E},B} v$ holds[2] between two Σ-terms u and v iff there is a position p, a rule $(t \to t') \in \vec{E}$ and a substitution θ such that $u|_p =_B t\theta$ and $v = u[t'\theta]_p$. Such decidability follows from the following theorem:

Theorem 3 *(Church-Rosser Theorem) [15]. Let $\mathcal{R} = (\Sigma, B, \vec{E})$ be a decomposition of $(\Sigma, E \uplus B)$. Then we have an equivalence:*

$$E \uplus \vdash u = v \quad \Leftrightarrow \quad u!_{\vec{E},B} =_B v!_{\vec{E},B}.$$

where $t!_{\vec{E},B}$ denotes the canonical form of term t by rewriting with $\to_{\vec{E},B}$, which exists and is unique up to B-equality thanks to the convergence of $\to_{\vec{E},B}$.

If $\mathcal{R} = (\Sigma, B, \vec{E})$ is a decomposition of $(\Sigma, E \uplus B)$ and X an S-sorted set of variables, the *canonical term algebra* $C_{\Sigma/\vec{E},B}(X)$ has $C_{\Sigma/\vec{E},B}(X)_s = \{[t!_{\vec{E},B}]_B \mid t \in T_\Sigma(X)_s\}$, and interprets each $f : s_1 \dots s_n \to s$ as the function $f_{C_{\Sigma/\vec{E},B}(X)} : ([u_1]_B, \dots, [u_n]_B) \mapsto [f(u_1, \dots, u_n)!_{\vec{E},B}]_B$. By the Church-Rosser Theorem we then have an isomorphism $h : T_{\Sigma/E}(X) \cong C_{\Sigma/\vec{E},B}(X)$, where $h : [t]_E \mapsto [t!_{\vec{E},B}]_B$. In particular, when X is the empty family of variables, the canonical term algebra $C_{\Sigma/\vec{E},B}$ is an initial algebra, and is the most intuitive model for $T_{\Sigma/E \uplus B}$ as an algebra of *values* computed by $\vec{E}, B$-simplification.

Quite often, the signature Σ on which $T_{\Sigma/E \uplus B}$ is defined has a natural decomposition as a disjoint union $\Sigma = \Omega \uplus \Delta$, where the elements of $C_{\Sigma/\vec{E},B}$ are Ω-terms, whereas the function symbols $f \in \Delta$ are viewed as *defined functions* which are *evaluated away* by $\vec{E}, B$-simplification. Ω (with same poset of

and substitutions ρ; and (ii) the axioms U oriented as rules $\vec{U}$ are *sort-decreasing* in the sense that $u = v \in U \Rightarrow ls(u\rho) \geqslant ls(v\rho)$ for each ρ. Maude automatically checks B-preregularity of an OS signature Σ in this broader sense [4].

[2] See [24] for the more general definition of both convergence and the relation $\to_{\vec{E},B}$ when Σ is B-preregular in the broader sense of Footnote 1.

sorts as Σ) is then called a *constructor subsignature* of Σ. Call a decomposition $\mathcal{R} = (\Sigma, B, \vec{E})$ of $(\Sigma, E \uplus B)$ *sufficiently complete* with respect to the *constructor subsignature* Ω iff for each $t \in T_\Sigma$ we have $t!_{\vec{E},B} \in T_\Omega$. Sufficient completeness is closely related to *protecting* inclusions of decompositions.

Definition 3 *(Protecting, Constructor Decomposition).* *A decomposition* $\mathcal{R} = (\Sigma, B, \vec{E})$ protects *decomposition* $\mathcal{R}_0 = (\Sigma_0, B_0, \vec{E}_0)$ *iff* $\Sigma_0 \subseteq \Sigma$, $B_0 \subseteq B$, *and* $\vec{E}_0 \subseteq \vec{E}$, *and for all* $t, t' \in T_{\Sigma_0}(X)$ *we have: (i)* $t =_{B_0} t' \Leftrightarrow t =_B t'$, *(ii)* $t = t!_{\vec{E}_0,B_0} \Leftrightarrow t = t!_{\vec{E},B}$, *and (iii)* $C_{\Sigma_0/\vec{E}_0,B_0} \cong C_{\Sigma/\vec{E},B}|_{\Sigma_0}$.

$\mathcal{R}_\Omega = (\Omega, B_\Omega, \vec{E}_\Omega)$ *is a* constructor decomposition *of* $\mathcal{R} = (\Sigma, B, \vec{E})$ *iff* $\mathcal{R}$ *protects* $\mathcal{R}_\Omega$ *and* Σ *and* Ω *have the same poset of sorts, so that* $\mathcal{R}$ *is sufficiently complete with respect to* Ω. *Finally,* Ω *is called a subsignature of* free constructors modulo B_Ω *iff* $\vec{E}_\Omega = \emptyset$, *so that* $C_{\Omega/\vec{E}_\Omega,B_\Omega} = T_{\Omega/B_\Omega}$.

The notion of *variant* answers, in a sense, two questions: (i) how can we best describe symbolically the elements of $C_{\Sigma/\vec{E},B}(X)$ that are *reduced substitution instances* of a *pattern term* t? and (ii) given an original pattern t, how many other patterns do we need to "cover" all reduced instances of t in $C_\mathcal{R}(X)$?

Definition 4. *Given a decomposition* $\mathcal{R} = (\Sigma, B, \vec{E})$ *and a* Σ*-term* t, *a* variant *[5, 11] of* t *is a pair* (u, θ) *such that: (i)* $u =_B (t\theta)!_{\vec{E},B}$, *(ii)* $dom(\theta) = vars(t)$, *and (iii)* $\theta = \theta!_{\vec{E},B}$, *that is,* $x\theta = (x\theta)!_{\vec{E},B}$ *for all variables* x. (u, θ) *is called a* ground variant *iff, furthermore,* $u \in T_\Sigma$. *Note that if* (u, θ) *is a ground variant of some* t, *then* $[u]_B \in C_{\Sigma/\vec{E},B}$. *Given variants* (u, θ) *and* (v, γ) *of* t, (u, θ) *is called* more general *than* (v, γ), *denoted* $(u, \theta) \sqsupseteq_B (v, \gamma)$, *iff there is a substitution* ρ *such that: (i)* $(\theta\rho)|_{vars(t)} =_B \gamma$, *and (ii)* $u\rho =_B v$. *Let* $[\![t]\!]_{\vec{E},B} = \{(u_i, \theta_i) \mid i \in I\}$ *denote a* complete set of variants *of* t, *that is, a set of variants such that for any variant* (v, γ) *of* t *there is an* $i \in I$, *such that* $(u_i, \theta_i) \sqsupseteq_B (v, \gamma)$. *A decomposition* $\mathcal{R} = (\Sigma, B, \vec{E})$ *of* $(\Sigma, E \uplus B)$ *has the* finite variant property *[5] (FVP) iff for each* Σ*-term* t *there is a* finite *complete set of variants* $[\![t]\!]_{\vec{E},B} = \{(u_1, \theta_1), \ldots, (u_n, \theta_n)\}$.

If B has a finitary unification algorithm and $\mathcal{R} = (\Sigma, B, \vec{E})$ is FVP, then for any term t the finite set $[\![t]\!]_{\vec{E},B}$ of its variants can be computed by *folding variant narrowing* [11]. Maude 2.7.1 supports the computation of $[\![t]\!]_{\vec{E},B}$ for B a combination of associative and/or commutative and/or identity axioms.

If a decomposition $\mathcal{R} = (\Sigma, B, \vec{E})$ is FVP and protects a constructor decomposition $\mathcal{R}_\Omega = (\Omega, B_\Omega, \vec{E}_\Omega)$, the notion of *constructor variant* answers the following related question: given a pattern t what are the reduced instances of t which "cover" all reduced *ground* instances of t?

Definition 5 *(Constructor Variant) [21].* *Let* $\mathcal{R} = (\Sigma, B, \vec{E})$ *be a decomposition of* $(\Sigma, E \uplus B)$, *and let* $\mathcal{R}_\Omega = (\Omega, B_\Omega, \vec{E}_\Omega)$ *be a constructor decomposition of* $\mathcal{R}$. *Then an* $\vec{E}, B$*-variant* (u, θ) *of a* Σ*-term* t *is called a* constructor $\vec{E}, B$-variant *of* t *iff* $u \in T_\Omega(X)$. *Let* $[\![t]\!]^\Omega_{\vec{E},B}$ *denote a complete set of constructor variants of*

a term t, i.e., for each constructor variant (v,β) of t there is a $(w,\alpha) \in [\![t]\!]^{\Omega}_{\vec{E},B}$ such that $(w,\alpha) \sqsupseteq_B (v,\beta)$.

Under mild conditions on a constructor decomposition $\mathcal{R}_\Omega = (\Omega, B_\Omega, \vec{E}_\Omega)$ protected by an FVP $\mathcal{R} = (\Sigma, B, \vec{E})$, if B has a finitary unification algorithm the set $[\![t]\!]^{\Omega}_{\vec{E},B}$ is finite and can be effectively computed according to the algorithm in [24], which has been implemented in Maude. Both the sets $[\![t]\!]_{\vec{E},B}$ and $[\![t]\!]^{\Omega}_{\vec{E},B}$ will play a key role in the various notions of ground coherence completion of a generalized rewrite theory presented in Sect. 4.

3 Generalized Rewrite Theories and Coherence

There are two main reasons for further generalizing the notion of rewrite theory in [3], and for relaxing its *executability conditions* as specified in, e.g., [8]. The first is that it has proved very useful to model *open systems* that interact with a typically non-deterministic external environment by rewrite rules that have extra variables in their righthand sides, so that a term t may be rewritten to a possibly *infinite* number of righthand side instances by different instantiations of such extra variables. The second reason is that for symbolic reasoning it is very useful to allow conditional rewrite rules $l \to r$ *if* φ where φ is not just a conjunction of equalities but a QF equational formula, which is viewed as a *constraint* imposed by the rule and interpreted in a suitable *background theory* T. The key point is that the notion of generalized rewrite theory thus obtained, although in general not executable in the standard sense, can still be executed *symbolically* under fairly reasonable assumptions. For example, the notion of *rewriting modulo SMT* [23] (see also the related work [1]) shows how such generalized theories can be symbolically executed under some typing restrictions and the requirement that satisfiability of a rule's condition φ is always decidable. Related, yet different, notions of symbolic execution for rules of this kind are also given in [12,16].

The purpose of this section is fourfold: (1) to give a general definition of such generalized rewrite theories with no executability or decidability assumptions at all; (2) to define a category of *transition system* models for generalized rewrite theories; (3) to first add executability assumptions to the equations in such theories; and (4) to then extend the notion of *coherence* [8,28] to generalized rewrite theories. This will have two important consequences: (i) it will provide essential conditions for *symbolic execution* of such generalized rewrite theories; and (ii) it will make the notion of *ground coherence completion* of a generalized rewrite theory presented in Sect. 4 as widely applicable as possible.

Definition 6 *(Generalized Rewrite Theory)***.** *A* generalized rewrite theory *is a 5-tuple $\mathcal{R} = (\Sigma, G, R, T, \phi)$, where: (i) Σ is* kind-complete*, so that its set of sorts is $S \uplus \widehat{S}$, (see Definition 1); (ii) (Σ, G) is a (possibly conditional) equational theory; (iii) R is a set of (possibly conditional) Σ-*rewrite rules*, i.e., sequents $l \to r$ if φ, with $l, r \in T_\Sigma(X)_{[s]}$ for some $[s] \in \widehat{S}$, and φ a QF Σ-formula; (iv) T, called the* background theory*, satisfies $(\Sigma, G) \subseteq T \subseteq th(T_{\Sigma/G})$; and (v)*

ϕ is a so-called frozenness *function,*[3] *mapping each subsort-polymorphic family $f_{[s]}^{[s_1]\ldots[s_n]}$ in Σ to the subset $\phi(f_{[s]}^{[s_1]\ldots[s_n]}) \subseteq \{1,\ldots,n\}$ of its* frozen arguments*.*

*Given a generalized rewrite theory $\mathcal{R} = (\Sigma, G, R, T, \phi)$ and terms $u, v \in T_{\Sigma,[s]}(X)$ for some $[s] \in \widehat{S}$, the rewrite relation $\rightarrow_{\mathcal{R}}$ holds between them, denoted $u \rightarrow_{\mathcal{R}} v$, iff there exist a term u', a ϕ-*unfrozen[4] *position p in u', a rule $l \rightarrow r$ if φ in R and a substitution θ such that: (i) $T \models \varphi\theta$; (ii) $u =_G u' = u'[l\theta]_p$; and (iii) $u'[r\theta]_p =_G v$.*

A generalized rewrite theory $\mathcal{R} = (\Sigma, G, R, T, \phi)$ is called topmost *iff there is a kind $[State] \in \widehat{S}$ such that: (i) for each $l \rightarrow r$ if φ in R, $l, r \in T_{\Sigma}(X)_{[State]}$; and (ii) for each subsort-polymorphic family $f_{[s]}^{[s_1]\ldots[s_n]}$ in Σ and $i \in \{1,\ldots,n\}$, if $[s_i] = [State]$, then $i \in \phi(f_{[s]}^{[s_1]\ldots[s_n]})$. For $\mathcal{R}$ topmost $u \rightarrow_{\mathcal{R}} v \Rightarrow u, v \in T_{\Sigma,[State]}$.*

Call $\mathcal{R} = (\Sigma, G, R, T, \phi)$ and $\mathcal{R}' = (\Sigma, G', R', T', \phi)$ semantically equivalent*, denoted $\mathcal{R} \equiv \mathcal{R}'$ (resp.* ground semantically equivalent*, denoted $\mathcal{R} \equiv_{gr} \mathcal{R}'$) iff: (1) $(\Sigma, G) \equiv (\Sigma, G')$, (2) $T \equiv T'$, and (3) $\rightarrow_{\mathcal{R}} = \rightarrow_{\mathcal{R}'}$ (resp. (1) $T_{\Sigma/G} = T_{\Sigma/G'}$, (2) $T \equiv T'$, and (3) $\rightarrow_{\mathcal{R}}|_{T_{\Sigma}^2} = \rightarrow_{\mathcal{R}'}|_{T_{\Sigma}^2}$).*

Note that the case of a *standard rewrite theory* is the special case where $\mathcal{R} = (\Sigma, G, R, T, \phi)$ is such that $T = (\Sigma, G)$ and for each $l \rightarrow r$ *if* φ in R, φ is a conjunction of equalities[5] $\varphi = \bigwedge_{i=1\ldots n} u_i = v_i$. In such a special case we omit the background theory and write $\mathcal{R} = (\Sigma, G, R, \phi)$ as usual. Note also that the QF formulas φ in the conditions of rules in R may not be arbitrary Σ-formulas, but formulas in a theory $T_0 = (\Sigma_0, \Gamma_0)$ such that $\Sigma_0 \subseteq \Sigma$. For example, T_0 may be the theory of Presburger arithmetic. In such a case, the background theory T in $\mathcal{R} = (\Sigma, G, R, T, \phi)$ is assumed to be a *conservative extension* of T_0.

Example 1. This QLOCK protocol example is borrowed from [25], where it is used to verify some of its properties in Reachability Logic by symbolic methods. It illustrates the new features of generalized rewrite theories, including a background theory, negative constraints in conditions, and "open system" rules modeling interaction with an outside environment. QLOCK can be formalized as a generalized rewrite theory $\mathcal{R} = (\widehat{\Sigma}, E \uplus B, R, th(T_{\widehat{\Sigma}/E \uplus B}), \phi)$, in the sense of Definition 6, where ϕ maps each $f \in \widehat{\Sigma}$ to $\emptyset$ (no frozen positions), and $\widehat{\Sigma}$ is the kind completion of signature Σ below. $\mathcal{R}$ models a dynamic version of the QLOCK

[3] This is supported in Maude by the `frozen` operator attribute, which forbids rewrites below the specified argument positions. For example, when giving a rewriting semantics to a CCS-like process calculus, the process concatenation operator $_\cdot_$, appearing in process expressions like $a \cdot P$, will typically be frozen in its second argument.

[4] By definition this means that there is no function symbol f and position q such that: (i) $p = q \cdot i \cdot q'$, (ii) $u'|_q = f(u_1,\ldots,u_n)$, and (iii) $i \in \phi(f_{[s]}^{[s_1]\ldots[s_n]})$. Intuitively this means that the frozenness restrictions ϕ do not block rewriting at position p in u'.

[5] Admittedly, it is possible to allow more general rules with additional "rewrite conditions" of the form $l \rightarrow r$ *if* $\varphi \wedge \bigwedge_{i=1\ldots n} u_i \rightarrow v_i$ in a generalized rewrite theory. Then, generalized rewrite theories would specialize to standard rewrite theories whose rules also allow rewrite conditions. I leave this further generalization as future work.

mutual exclusion protocol [13], where (Σ, B) defines the protocol's states, involving natural numbers, lists, and multisets over natural numbers. Σ has sorts $S = \{Nat, List, MSet, Conf, State, Pred\}$ with subsorts $Nat < List$ and $Nat < MSet$ and operators $F = \{0 : \to Nat,\ s_ : Nat \to Nat,\ \emptyset : \to MSet,\ nil : \to List,\ __ : MSet\ MSet \to MSet,\ _;_ : List\ List \to List, dupl : MSet \to Pred, tt : \to Pred, _|_|_|_ : MSet\ MSet\ MSet\ List \to Conf, <_> : Conf \to State\}$, where underscores denote operator argument placement. The axioms B are the associativity-commutativity of the multiset union $__$ with identity $\emptyset$, and the associativity of list concatenation $_;_$ with identity nil. The only equation in E is $dupl(s\,i\,i) = tt$. It defines the *dupl* predicate by detecting a duplicated element i in the multiset $s\,i\,i$ (where s could be empty). The *states* of QLOCK are B-equivalence classes of ground terms of sort *State*.

QLOCK [13] is a mutual exclusion protocol where the number of processes is unbounded. Furthermore, in the *dynamic* version of QLOCK presented below, such a number can grow or shrink. Each process is identified by a number. The system configuration has three sets of processes (normal, waiting, and critical) plus a waiting queue. To ensure mutual exclusion, a normal process must first register its name at the end of the waiting queue. When its name appears at the front of the queue, it is allowed to enter the critical section. The first three rewrite rules in R below specify how a *normal* process i first transitions to a *waiting* process, then to a *critical* process, and back to normal. The last two rules in R specify how a process can dynamically join or exit the system.

$$\begin{array}{llllllllll}
n2w : & < n\ i & \mid w & \mid c & \mid q & > \;\to & < n & \mid w\ i & \mid c & \mid q\ ;\ i > \\
w2c : & < n & \mid w\ i & \mid c & \mid i\ ;\ q & > \;\to & < n & \mid w & \mid c\ i & \mid i\ ;\ q > \\
c2n : & < n & \mid w & \mid c\ i & \mid i\ ;\ q & > \;\to & < n\ i & \mid w & \mid c & \mid q > \\
join : & < n & \mid w & \mid c & \mid q & > \;\to & < n\ i & \mid w & \mid c & \mid q > \ if\ \varphi \\
exit : & < n\ i & \mid w & \mid c & \mid q & > \;\to & < n & \mid w & \mid c & \mid q >
\end{array}$$

where $\varphi \equiv dupl(n\,i\,w\,c) \neq tt$, i is a number, n, w, and c are, respectively, normal, waiting, and critical process identifier sets, and q is a queue of process identifiers. Note that *join* makes QLOCK an *open* system in the sense explained earlier in this section. In the intended use of QLOCK, any state $< n \mid w \mid c \mid q >$ will be such that the multiset $n\,w\,c$ is actually a *set*, so that $dupl(n\,w\ c) \neq tt$ holds. Note that this is an invariant preserved by all the above rules.

Transition System Semantics of Generalized Rewrite Theories. Given a generalized rewrite theory $\mathcal{R} = (\Sigma, G, R, T, \phi)$ we can associate to it the transition system $\mathcal{T}_{\mathcal{R}} = (T_{\Sigma/G}, \to_{\mathcal{R}})$, resp. $\mathcal{T}_{\mathcal{R}}(X) = (T_{\Sigma/G}(X), \to_{\mathcal{R}})$, where, by definition, given $[u], [v] \in T_{\Sigma/G,[s]}$ (resp. $[u], [v] \in T_{\Sigma/G,[s]}(X)$) for some $[s] \in \widehat{S}$, $[u] \to_{\mathcal{R}} [v]$ holds iff $u \to_{\mathcal{R}} v$ holds in the sense of Definition 6. Both $\mathcal{T}_{\mathcal{R}}$ and $\mathcal{T}_{\mathcal{R}}(X)$ are Σ-transition system in the following sense:

Definition 7 *(Σ-Transition System and Homomorphism)***.** *Given a kind-complete OS signature Σ, a Σ*-transition system *is a pair $(A, \to_A)$ where: (i) A is a Σ-algebra; and (ii) $\to_A$ is a $\widehat{S}$-indexed family of relations $\to_A = \{\to_{A_{[s]}} \subseteq A^2_{[s]}\}_{[s]\in\widehat{S}}$.*

A homomorphism *of Σ-transition systems $h : (A, \rightarrow_A) \rightarrow (B, \rightarrow_B)$ is a Σ-homomorphism $h : A \rightarrow B$ such that for each $[s] \in \widehat{S}$ and $a, a' \in A_{[s]}$, $a \rightarrow_{A_{[s]}} a'$ implies $h(a) \rightarrow_{B_{[s]}} h(a')$. This defines a category* $\mathbf{Trans}_\Sigma$.

Note that $h : (A, \rightarrow_A) \rightarrow (B, \rightarrow_B)$ is an isomorphism in this category iff: (i) h is a Σ-isomorphism, and (ii) $b \rightarrow_{B_{[s]}} b'$ implies $h^{-1}(b) \rightarrow_{A_{[s]}} h^{-1}(b')$. Intuitively, such an isomorphism could be called an "algebraic bisimulation," and a homomorphism an "algebraic simulation."

Given a generalized rewrite theory $\mathcal{R} = (\Sigma, G, R, T, \phi)$ we say that a Σ-transition system $(A, \rightarrow_A)$ *satisfies* the theory $\mathcal{R}$, denoted $(A, \rightarrow_A) \models \mathcal{R}$ iff: (i) $A \in \mathbf{OSAlg}_{(\Sigma,G)}$, and (ii) for each $\alpha \in [Y \rightarrow A]$ the unique Σ-homomorphism $_\alpha : T_{\Sigma/G}(X) \rightarrow A$ is a Σ-transition system homomorphism $_\alpha : \mathcal{T}_\mathcal{R}(X) \rightarrow (A, \rightarrow_A)$. This defines a full subcategory $\mathbf{Trans}_\mathcal{R} \subseteq \mathbf{Trans}_\Sigma$ whose initial object is $\mathcal{T}_\mathcal{R}$. When $\mathcal{R} = (\Sigma, G, R, \phi)$ is a standard rewrite theory, the Σ-transition system $\mathcal{T}_\mathcal{R}$ is closely related to the *initial reachability model* of $\mathcal{R}$ [3], whose associated Σ-transition system is the transitive closure $(T_{\Sigma/G}, \rightarrow^*_\mathcal{R})$ of $\mathcal{T}_\mathcal{R}$. Roughly speaking, $\mathcal{T}_\mathcal{R}$ is the "one step rewrite" fragment of the initial reachability model in [3].

Definition 6 is very general: in $\mathcal{R} = (\Sigma, G, R, T, \phi)$, besides the generality of the rules R, no assumptions are made about the (possibly conditional) equations G which we are rewriting *modulo* in each transition $u \rightarrow_\mathcal{R} v$. In such generality, even *symbolic* execution of $\mathcal{R}$ may be hard to attain. We can substantially improve the situation if we assume that $G = E \uplus B$, with B regular and linear unconditional axioms for which Σ is B-preregular and $=_B$ is decidable, and such that (Σ, G) has a *decomposition* $(\Sigma, B, \vec{E})$. Strictly speaking, such decompositions have only been defined in Sect. 2 for G a set of *unconditional* equations. However, as shown in, e.g., [8,18], the notion of decomposition of $(\Sigma, E \uplus B)$ generalizes to conditional equations E by means of the notion of a *convergent, strongly deterministic* rewrite theory $(\Sigma, B, \vec{E})$. Likewise, the Church-Rosser Theorem, the notion of canonical term algebra $C_{\Sigma/\vec{E},B}$, and the isomorphism $C_{\Sigma/E,B} \cong T_{\Sigma/E \uplus B}$ naturally extend to the conditional case for such decompositions [18]. Under such conditions, we can achieve a much simpler rewrite relation $\rightarrow_{R/B}$ with the rules R modulo B. Given two terms $u, v \in T_{\Sigma,[s]}(X)$ for some $[s] \in \widehat{S}$, the rewrite relation $u \rightarrow_{R/B} v$ holds iff there exists a $u' \in T_\Sigma(X)$ with $u =_B u'$, a ϕ-*unfrozen* position p in u', a rule $l \rightarrow r$ *if* φ in R and a substitution θ such that: (i) $T \models \varphi\theta$; (ii) $u'|_p = l\theta$; and (iii) $v = u'[r\theta]_p$. Under these extra assumptions on $\mathcal{R}$, much simpler Σ-transition systems can be defined:

Definition 8 *(Canonical Σ-Transition System). Let $\mathcal{R} = (\Sigma, E \uplus B, R, T, \phi)$ be such that $(\Sigma, E \uplus B)$ has a decomposition $(\Sigma, B, \vec{E})$ in the above-mentioned sense. Then the Σ-transition system $\mathcal{C}_\mathcal{R}(X)$ (resp. $\mathcal{C}_\mathcal{R}$) is defined as the pair $(C_{\Sigma/\vec{E},B}(X), \rightarrow_{\mathcal{C}_\mathcal{R}})$ (resp. $(C_{\Sigma/\vec{E},B}, \rightarrow_{\mathcal{C}_\mathcal{R}} |_{C^2_{\Sigma/\vec{E},B}})$) where for $[u], [v] \in C_{\Sigma/\vec{E},B}(X)$ (resp. $[u], [v] \in C_{\Sigma/\vec{E},B}$), $[u] \rightarrow_{\mathcal{C}_\mathcal{R}} [v]$ holds iff there exists $w \in T_\Sigma(X)$ such that: (i) $u \rightarrow_{R/B} w$, and (ii) $[v] = [w!_{\vec{E},B}]$.*

The Coherence Problem. Note that it follows from the above definition and from Definition 6 that if $[u]_B \rightarrow_{\mathcal{C}_\mathcal{R}} [v]_B$, then $[u]_{E \uplus B} \rightarrow_\mathcal{R} [v]_{E \uplus B}$. And since

the isomorphism $h : C_{\Sigma/\vec{E},B} \cong T_{\Sigma/E \uplus B}$ (resp. $h : C_{\Sigma/\vec{E},B}(X) \cong T_{\Sigma/E \uplus B}(X)$) is precisely the mapping $h : [u]_B \mapsto [u]_{E \uplus B}$, this means that we have a homomorphism of Σ-transition systems $h : \mathcal{C}_{\mathcal{R}} \to \mathcal{T}_{\mathcal{R}}$ (resp. $h : \mathcal{C}_{\mathcal{R}}(X) \to \mathcal{T}_{\mathcal{R}}(X)$). However, although h is a Σ-isomorphism, it fails in general to be an isomorphism of Σ-transition systems. This is well-known for even trivially simple rewrite theories $\mathcal{R} = (\Sigma, E \uplus B, R, \phi)$ such as $\mathcal{R}$ with Σ unsorted and consisting of constants a, b, c, $E = \{a = b\}$, $B = \emptyset$, and $R = \{a \to c\}$, where $\to_{\mathcal{C}_{\mathcal{R}}} = \emptyset$, but $\to_{\mathcal{R}} = \{(\{a, b\}, \{c\})\}$. Since $\mathcal{T}_{\mathcal{R}}$ is initial in $\mathbf{Trans}_{\mathcal{R}}$, this of course means that in general $\mathcal{C}_{\mathcal{R}} \notin \mathbf{Trans}_{\mathcal{R}}$, and likewise $\mathcal{C}_{\mathcal{R}}(X) \notin \mathbf{Trans}_{\mathcal{R}}$. Therefore, canonical transition systems, although simpler than $\mathcal{T}_{\mathcal{R}}$ or $\mathcal{T}_{\mathcal{R}}(X)$, *cannot* be used to reason correctly about $\mathcal{R}$-computations. This is the so-called *coherence problem.*

Call $\mathcal{R} = (\Sigma, E \uplus B, R, T, \phi)$ with decomposition $(\Sigma, B, \vec{E})$ *coherent* (resp. *ground coherent*) iff the Σ-transition system homomorphism $h : \mathcal{C}_{\mathcal{R}}(X) \to \mathcal{T}_{\mathcal{R}}(X)$ (resp. $h : \mathcal{C}_{\mathcal{R}} \to \mathcal{T}_{\mathcal{R}}$) is an isomorphism. Coherence can be characterized by an easier to check condition that generalizes ideas in [8, 28]:

Theorem 4. *Let $\mathcal{R} = (\Sigma, E \uplus B, R, T, \phi)$ with $(\Sigma, E \uplus B)$ a decomposition of $(\Sigma, B, \vec{E})$. Then $\mathcal{R}$ is coherent (resp. ground coherent) iff for each $u, v \in T_\Sigma(X)$ (resp. $u \in T_\Sigma$, $v \in T_\Sigma(X)$) such that $u \to_{R/B} v$ (resp. $u \to_{R/B} v$ and $v!_{\vec{E},B} \in T_\Sigma$) there is a term $v' \in T_\Sigma(X)$ such that $u!_{\vec{E},B} \to_{R/B} v'$ and $v!_{\vec{E},B} =_B v'!_{\vec{E},B}$.*

The methods developed in [8] to *check* the coherence of a given $\mathcal{R}$ are based on adequate *critical pairs* modulo B between conditional rules in R and (oriented) conditional equations in $\vec{E}$. By generalizing the conditions in [8] from conjunctions of equalities to QF equational formulas and dropping the executability conditions in [8], general methods for *coherence checking* entirely similar to those in [8] can be developed for generalized rewrite theories. This, however, is *not* the focus of this paper. Instead, both for the special case of the rewrite theories in [8] and for the generalized rewrite theories in Definition 6 above, a *different* question is asked and answered for the first time: Can we, under suitable conditions, *transform* a generalized rewrite theory $\mathcal{R}$ into a semantically equivalent theory $\overline{\mathcal{R}}$, called its *ground coherence completion*, so that $\overline{\mathcal{R}}$ is itself ground coherent? This question is answered in Sect. 4 below.

4 Coherence Completion of Generalized Rewrite Theories

I present below several theory transformations making a given generalized rewrite theory ground coherent. I also explain how these methods can be automated and how they can be applied to: (i) make rewrite theories symbolically executable; (ii) reason about *equational abstractions* of rewrite theories [22], and (iii) achieve symbolic execution of a widest possible class of such rewrite theories. But first some assumptions on $\mathcal{R}$ need to be made.

Assumptions on $\mathcal{R}$. The generalized rewrite theory $\mathcal{R}$ has the form $\mathcal{R} = (\Sigma, E \uplus B, R, T, \phi)$, with $(\Sigma, E \uplus B)$ a decomposition of $(\Sigma, B, \vec{E})$. Furthermore: (i) $\mathcal{R}$ is *topmost*; (ii) there are protecting inclusions of decompositions[6]

$$(\Omega, B_\Omega, \vec{E}_\Omega) \subseteq (\Sigma_1, B_1, \vec{E_1}) \subseteq (\Sigma, B, \vec{E})$$

where: (a) Ω, Σ_1 and Σ share the same poset of sorts; (b) E_Ω and E_1 are *unconditional* equations; (c) $(\Omega, B_\Omega, \vec{E}_\Omega)$ is a *constructor* decomposition of $(\Sigma, B, \vec{E})$ and, a fortiori, of $(\Sigma_1, B_1, \vec{E_1})$; and (d) $(\Sigma_1, B_1, \vec{E_1})$ is an FVP decomposition; and (iii) each rewrite rule $l \to r \text{ } if \text{ } \varphi$ in R is such that l is a Σ_1-term.

Are these assumptions "reasonable"? Regarding assumption (i), many rewrite theories of interest, including theories specifying distributed object-oriented systems and rewriting logic specifications of concurrent programming languages, can be easily specified as topmost rewrite theories by simple theory transformations, e.g., [27]. Regarding assumption (ii)–(iii), some remarks are in order. First, the specification of a constructor subsignature Ω is either explicit in most applications or typically easy to carry out. Second, in virtually all practical specifications of rewrite theories the lefthand side l of a rule $l \to r \text{ } if \text{ } \varphi$ is almost always a constructor term. The only case in which this may happen to fail in practice is the case of an *equational abstraction* [22], where l typically *was* a constructor term *before* the abstraction was defined, but after such abstraction definition a *smaller* signature Ω of constructors can be defined. This means that for some applications the decomposition $(\Sigma_1, B_1, \vec{E_1})$ may specify an equational abstraction.

However, $\mathcal{R}$ need *not* be an equational abstraction of another rewrite theory. The FVP decomposition $(\Sigma_1, B_1, \vec{E_1})$ may have other meanings, including $(\Sigma_1, B_1, \vec{E_1}) = (\Omega, B_\Omega, \vec{E}_\Omega)$, so that the general assumptions are not at all restricted to equational abstractions. This will become clear in what follows.

The $\mathcal{R} \mapsto \overline{\mathcal{R}}_l$ Transformation. For $\mathcal{R} = (\Sigma, E \uplus B, R, T, \phi)$ satisfying the above assumptions, the theory $\overline{\mathcal{R}}_l$ has the form $\overline{\mathcal{R}}_l = (\Sigma, E \uplus B, \overline{R}_l, T, \phi)$, where

$$\overline{R}_l = \{l' \to (r\gamma)!_{\vec{E},B} \text{ } if \text{ } (\varphi\gamma)!_{\vec{E},B} \mid (l', \gamma) \in [\![l]\!]_{\vec{E_1},B_1} \wedge l \to r \text{ } if \text{ } \varphi \in R\}.$$

As an optimization, we can remove from $\overline{\mathcal{R}}_l$ those rules B-*subsumed* by other rules in $\overline{\mathcal{R}}_l$, where the B subsumption relation $(l \to r \text{ } if \text{ } \varphi) \sqsupseteq_B (l' \to r' \text{ } if \text{ } \varphi')$ holds between rules iff there is a substitution α such that $l\alpha =_B l'$, $r\alpha =_B r'$ and $\varphi\alpha =_B \varphi'$. That is, $l \to r \text{ } if \text{ } \varphi$ is *more general* than $l' \to r' \text{ } if \text{ } \varphi'$ up to B-equality, making $l' \to r' \text{ } if \text{ } \varphi'$ redundant. The transformation $\mathcal{R} \mapsto \overline{\mathcal{R}}_l$ can be easily automated as a meta-level function in Maude 2.7.1 using the `metaGetIrredundantVariant` function.

Theorem 5. *Under the above assumptions on $\mathcal{R}$, $\overline{\mathcal{R}}_l$ is semantically equivalent to $\mathcal{R}$. Furthermore, $\overline{\mathcal{R}}_l$ is ground coherent.*

[6] Recall that the strongly deterministic and convergent rules $\vec{E}$ may be *conditional*. We are therefore using Definition 3 in its straightforward generalization to the conditional case.

Example 2. The $\mathcal{R} \mapsto \overline{\mathcal{R}}_l$ transformation can be used to obtain a ground coherent theory for an equational abstraction of an infinite-state, out-of-order and fault-tolerant communication channel, which thus becomes finite-state and therefore analyzable by standard LTL model checking. Full details are given in [20]. Here I illustrate the transformation by focusing on one of the rules, namely, the message reception rule:

```
rl [recv] : [L,N] {J,K} S [P,M] =>
              [K ~ M, [L,N] S ack(K) [P ; J, M + 1],
                      [L,N] S ack(K) [P,M]] .
```

The rule's lefthand side describes a state in which the sender's state `[L,N]` consists of a list `L` of items still to be sent, and a counter `N`, and the receiver's state `[P,M]` consists of a list `P` of items already received and a counter `M`. The channel's contents is a multiset of messages with multiset union denoted by juxtaposion. In this case the contents of the channel is the multiset `{J,K} S` where `{J,K}` is a message sending item `J` marked as message number number `K` sent by the sender to ensure in-order communication. The rest of the messages in the channel are described by the variable `S` of sort `MsgSet`. The rule's righthand side describes two alternative behaviors of the receiver by means of an if-then-else operator

```
op [_,_,_] : Bool Channel Channel -> Channel [frozen] .
```

which is declared `frozen` so that no further rewrites below it are possible until after the if-then-else has been evaluated away. Depending on the equality test `K ~ M` between the message number `K` in the message and the receiver's counter `M`, the sender either appends the item at the end of its list and increases its counter, or discards the message without changing its counter. But in either case an `ack(K)` message signaling the receipt of message number `K` is sent to the sender.

Besides the *associativity* axiom for the list concatenation operator `_;_` and the *associativity-commutativity* axioms for the multiset union operator `_ _` plus the usual equations for if-then-else and the number equality predicate, the key equations in this module are:

```
eq L ; nil = L [variant] .
eq nil ; L = L [variant] .
eq L ; nil ; Q = L ; Q [variant] .  *** B-coherence extension

eq S null = S [variant] .
eq S S = S [variant] .
eq S S S' = S S' [variant] .     *** B-coherence extension
```

The first three equations make `nil` an identity element for list concatenation. The fourth equation likewise makes `null` an identity element for multiset union. With these equations alone the system is *infinite-state* due to the possibility of message loss modeled by the conditional rule

```
crl [loss] : [L,N] S S' [P,M] => [L,N] S' [P,M] if S =/= null .
```

which makes the specification into a *generalized* rewrite theory due to its QF negative condition. Message loss forces the sender to keep resending each item by means of a `[send]` rule not presented here. The system is made *finite-state*, and therefore verifiable by standard LTL model checking, by means of the equational abstraction [22] provided by the last two idempotency equations, because the unbounded multiset of messages in the channel thus becomes a set of bounded size. All equations involved are FVP so that the requirements in Theorem 5 are met. For $\mathcal{R}$ the generalized rewrite theory specifying this equationally-abstracted channel, its ground coherence completion $\mathcal{R} \mapsto \overline{\mathcal{R}}_l$ is described in full detail in [20]. Here we can just get a flavor for this theory transformation by focusing on the "variants" of the above `[recv]` rule which are added, namely, the following rules:

```
rl [recv] : [L,N] {J,K} [P,M] =>
              [(K ~ M), [L,N] ack(K) [P ; J, M + 1],
                      [L,N] ack(K) [P,M]] .
rl [recv] : [L,N] {J,K} [P,M] =>
              [K ~ M, [L,N] {J,K} ack(K) [P ; J, M + 1],
                      [L,N] {J,K} ack(K) [P,M]] .
rl [recv] : [L,N] {J,K} S [P,M] =>
              [K ~ M, [L,N] {J,K} S ack(K) [P ; J, M + 1],
                      [L,N] {J,K} S ack(K) [P,M]] .
```

The $\mathcal{R} \mapsto \mathcal{R}_{\Sigma_1}$ Transformation. The transformation $\mathcal{R} \mapsto \mathcal{R}_{\Sigma_1}$ is not a coherence completion, but a stepping stone towards a more powerful such completion discussed later. The problem solved by the transformation $\mathcal{R} \mapsto \mathcal{R}_{\Sigma_1}$ has everything to do with symbolic execution and is the following. As already mentioned, a generalized rewrite theory $\mathcal{R}$ of practical interest will typically have rules $l \rightarrow r$ *if* φ where the lefthand side l is either a constructor term, or at least a Σ_1-term with $(\Sigma_1, B_1, \vec{E_1})$ FVP. But what about the rule's righthand side r? Nothing can be assumed in general about r. It can be an arbitrary Σ-term because *auxiliary functions* in Σ may be needed to *update* the state. This poses a serious challenge for *symbolic reasoning* about $\mathcal{R}$, which typically will use symbolic methods such as equational unification and reachability analysis by narrowing modulo an equational theory. As long as r is an Ω-term or at least a Σ_1-term with $(\Sigma_1, B_1, \vec{E_1})$ FVP, this can easily be done *after each symbolic transition step*, because we can use variant-based unification to compute unifiers in the FVP theories $(\Omega, B_\Omega, \vec{E}_\Omega)$ or $(\Sigma_1, B_1, \vec{E_1})$, and likewise narrowing modulo such theories to perform symbolic reachability analysis. Instead, if, as usual, r is an arbitrary Σ-term, symbolic reasoning, while not impossible, becomes much harder: if the decomposition $(\Sigma, B, \vec{E})$ is unconditional, we can still perform variant $E \uplus B$-unification by variant narrowing as supported in Maude 2.7.1 for convergent unconditional theories, and likewise narrowing-based reachability analysis based on such $E \uplus B$-unification; but the number of unifiers is in general *infinite*, leading to impractical search spaces with potentially infinite branching at each symbolic state. In Lenin's words: *what is to be done*? Perform the $\mathcal{R} \mapsto \mathcal{R}_{\Sigma_1}$ transformation! This transformation generalizes to a general

FVP decomposition $(\Sigma_1, B_1, \vec{E}_1)$ between $(\Omega, B_\Omega, \vec{E}_\Omega)$ and a possibly conditional $(\Sigma, B, \vec{E})$ the special case, described in [25], of a transformation $\mathcal{R} \mapsto \mathcal{R}_\Omega$ making all righthand sides constructor terms. The extra generality of $\mathcal{R} \mapsto \mathcal{R}_{\Sigma_1}$ is useful, because it has a better chance of becoming the identity transformation for many rules in $\mathcal{R}$. Note that, since righthand sides in $\mathcal{R}_{\Sigma_1}$ are Σ_1-terms, a rule $\alpha : l \rightarrow r \textit{ if } \varphi$ can be applied *backwards*, as the rule $\alpha^{-1} : r \rightarrow l \textit{ if } \varphi$, to perform *backwards* symbolic reachability analysis, as done in Maude-NPA [10].

The transformation $\mathcal{R} \mapsto \mathcal{R}_{\Sigma_1}$ is defined as follows. By our assumptions on $\mathcal{R}$ each rewrite rule has the form $l \rightarrow r \textit{ if } \varphi$ with $l \in T_{\Sigma_1}(X)$. For symbolic reasoning purposes it will be very useful to also achieve that $r \in T_{\Sigma_1}(X)$. If $\mathcal{R} = (\Sigma, E \cup B, R, T, \phi)$, $\mathcal{R}_{\Sigma_1}$ has the form $\mathcal{R}_{\Sigma_1} = (\Sigma, E \cup B, R_{\Sigma_1}, T, \phi)$, where the rules in R_{Σ_1} are obtained from those in R by transforming each $l \rightarrow r \textit{ if } \varphi$ in R into the rule $l \rightarrow r' \textit{ if } \varphi \wedge \hat{\theta}$, where: (i) $r' \in T_{\Sigma_1}(X)$ is the Σ_1-*abstraction* of r obtained by replacing each length-minimal position p of r where the *top symbol* $\textit{top}(t|_p)$ of $t|_p$ does not belong Σ_1 by a fresh variable x_p whose sort is the least sort of $t|_p$, and (ii) $\hat{\theta} = \bigwedge_{p \in P} t|_p = x_p$, where P is the set of all length-minimal positions in r with $\textit{top}(t|_p) \notin \Sigma_1$. As an optimization, whenever $p, p' \in P$ are such that $t_p =_B t'_p$, we can use the same fresh variable for x_p and $x_{p'}$.

Example 3. Since, by specifying order in the natural numbers with constructors an ACU addition $+$, constants $0, 1$ of sort $\textit{Nat}$, and $\top, \bot$ of sort $\textit{Bool}$, Presburger arithmetic with $>$ and $\geqslant$ predicates and extended also with an if-then-else operator $[_,_,_]$ added to any desired sort has an FVP decomposition with signature Σ_1 with decidable $\textit{th}(T_{\Sigma_1/E_1 \uplus B_1})$ [21], if we have a topmost system whose states are pairs $\langle n, m \rangle$ of natural numbers, and where one of its rules has the form:

$$\langle n, m \rangle \rightarrow [n > m, \langle n * m, m \rangle, \langle n, n * m \rangle]$$

then, since the multiplication operator $_ * _$ is in Σ but outside Σ_1, the set P of length-minimal positions of the righthand side is $P = \{2.1, 3.2\}$. And since the terms at such positions are both $n * m$, we obtain the transformed rule:

$$\langle n, m \rangle \rightarrow [n > m, \langle y, m \rangle, \langle n, y \rangle] \textit{ if } y := n * m.$$

where y has sort $\textit{Nat}$ and I have used Maude's "matching condition" notation $y := n*m$ for the equation $n*m = y$ to emphasize its executability by matching, which, operationally, corresponds to viewing it as an *equational* rewrite condition of the form $n * m \rightarrow^*_{\vec{E},B} y$.

Although a generalized rewrite theory $\mathcal{R}$ need not be executable, the $\mathcal{R} \mapsto \mathcal{R}_{\Sigma_1}$ transformation *preserves rule executability*. To explain this, I need to explain the general sense in which a rewrite rule $l \rightarrow r \textit{ if } \varphi$ in R with $\varphi = \bigwedge_{i=1..n} u_i = v_i$ a conjunction of equalities becomes *executable* by evaluating its condition φ by $\vec{E}, B$ rewriting and B-matching. The sense, as explained in [8], is that we view φ as a $\vec{E}, B$-rewrite condition $\bigwedge_{i=1..n} u_i \rightarrow v_i$ and require the following *strong determinism* conditions: (i) $\forall j \in [1..n], \textit{vars}(u_j) \subseteq \textit{vars}(l) \cup \bigcup_{k<j} \textit{vars}(v_k)$, (ii) $\textit{vars}(r) \subseteq \textit{vars}(l) \cup \bigcup_{j \leqslant n} \textit{vars}(v_j)$, and (iii) each v_j is *strongly* $\vec{E}, B$*-irreducible*

in the precise sense that $v_j\sigma$ is in $\vec{E},B$-normal form for each $\vec{E},B$-normalized substitution σ. The point is that if properties (i)–(ii) hold for the original rule $l \to r$ *if* φ in R, then they also hold for its transformed rule $l \to r'$ *if* $\varphi \wedge \hat{\theta}$ in R_{Σ_1}. This is clear for (i) and (ii) by construction, and follows also for (iii) because in each rewrite condition $t|_p \to x_p$ obtained from $\hat{\theta}$ the variable x_p is trivially strongly $\vec{E},B$-irreducible. In summary we have:

Theorem 6. *Under the above assumptions on* $\mathcal{R}$ *(dropping the topmost assumption),* $\mathcal{R}_{\Sigma_1}$ *is semantically equivalent to* $\mathcal{R}$*. Furthermore, if the rules in* $\mathcal{R}$ *are executable in the above sense, then those is* $\mathcal{R}_{\Sigma_1}$ *are also executable.*

The $\mathcal{R} \mapsto \overline{\mathcal{R}}^{\Omega}_{\Sigma_1,l,r}$ **Transformation.** We can now use the previous $\mathcal{R} \mapsto \mathcal{R}_{\Sigma_1}$ transformation to achieve *simultaneously* two important goals: (1) obtain a generalized rewrite theory $\overline{\mathcal{R}}^{\Omega}_{\Sigma_1,l,r}$ ground semantically equivalent to $\mathcal{R}$ and such that the lefthand and righthand sides of each of its rules are *constructor* terms; this can be very useful for symbolic executability purposes, since we only need to perform $E_\Omega \uplus B_\Omega$-unification steps, which in many examples may reduce to just B_Ω-unification steps; and (2) ensure that $\overline{\mathcal{R}}^{\Omega}_{\Sigma_1,l,r}$ is *ground coherent.*

As already mentioned, the transformation $\mathcal{Q} \mapsto \mathcal{Q}_{\Sigma_1}$ will be used here as a stepping stone. Therefore, we may assume without loss of generality that *it has already been applied*, so that the input theory in this, second transformation $\mathcal{R} \mapsto \overline{\mathcal{R}}^{\Omega}_{\Sigma_1,l,r}$ is of the form $\mathcal{R} = \mathcal{Q}_{\Sigma_1}$. Therefore, $\mathcal{R} = (\Sigma, E \cup B, R, T, \phi)$ is such that in each rule $l \to r$ *if* φ in R both l and r are Σ_1-terms, where $(\Sigma_1, B_1, \vec{E}_1)$ is an FVP decomposition protecting a constructor decomposition $(\Omega, B_\Omega, \vec{E}_\Omega)$ and itself protected by $(\Sigma, B, \vec{E})$. The transformed theory $\overline{\mathcal{R}}^{\Omega}_{\Sigma_1,l,r}$ has then the form $\overline{\mathcal{R}}^{\Omega}_{\Sigma_1,l,r} = (\Sigma, E \cup B, R^{\Omega}_{\Sigma_1,l,r}, T, \phi)$, where

$$R^{\Omega}_{\Sigma_1,l,r} = \{l' \to r' \textit{ if } (\varphi\gamma)!_{\vec{E},B} \mid (l \to r \textit{ if } \varphi) \in R \wedge (\langle l', r'\rangle, \gamma) \in [\![\langle l, r\rangle]\!]^{\Omega}_{\vec{E}_1,B_1}\}$$

where we assume without loss of generality that a pairing operator $\langle _, _ \rangle$ has been added as a free constructor to each kind in Σ_1 and therefore also to Ω. The key point, of course, is that now the lefthand and righthand sides of a rule $l' \to r'$ *if* $(\varphi\gamma)!_{\vec{E},B}$ in $R^{\Omega}_{\Sigma_1,l,r}$ are *constructor terms.* This has two important advantages: (1) such rules can be symbolically executed, for example for reachability analysis, by performing $E_\Omega \uplus B_\Omega$-unification, which it typically much simpler and efficient that $E_1 \uplus B_1$-unification; and (2) a rule $\alpha : l' \to r'$ *if* $(\varphi\gamma)!_{\vec{E},B}$ can be executed *backwards* as the rule $\alpha^{-1} : r' \to l'$ *if* $(\varphi\gamma)!_{\vec{E},B}$, which can be very useful for backwards symbolic reachability analysis. Here are the key properties:

Theorem 7. *Under the above assumptions on* $\mathcal{R}$*,* $\overline{\mathcal{R}}^{\Omega}_{\Sigma_1,l,r}$ *is ground semantically equivalent to* $\mathcal{R}$*. Furthermore,* $\overline{\mathcal{R}}^{\Omega}_{\Sigma_1,l,r}$ *is ground coherent.*

Example 4. The $\mathcal{R} \mapsto \overline{\mathcal{R}}^{\Omega}_{\Sigma_1,l,r}$ transformation can be illustrated by a bank account system which is an open system and uses various auxiliary functions to update an account's state after each transaction. Full details are given in [20]. Here I illustrate the transformation by focusing on one of the rewrite rules, namely, the rule `[w]` specifying how money can be withdrawn from an account:

```
rl [w] :  < bal: n pend: x overdraft: false > # withdraw(m),msgs =>
       [ m > n , < bal: n pend: x overdraft: true > # msgs ,
           < bal: (n - m) pend: (x - m) overdraft: false > # msgs ]   .
```

The rule's lefthand side describes the state of the account, which is a `#`-separated pair. The record `< bal: n pend: x overdraft: false >` is the first component. The `bal`ance `n` is the amount of money currently in the account, `x` is the amount of money `pend`ing to be withdrawn in the future, which can be thought of as the amount corresponding to previously written but not yet cashed checks and other withdrawals, and `overdraft` is a Boolean flag whose `false` value indicates that the account is not in the red. Its second component is a multiset of messages built up with an *associative-commutative* multiset union operator `_,_` with *identity* element the empty multiset `mt`. It models the checks and other withdrawals pending to be cashed. Here such a multiset has the form `withdraw(m),msgs` so that there is an actual request `withdraw(m)` to withdraw the amount of money `m` and the remaining messages described by the variable `msgs`. The rule's righthand side describes the account's behavior in response to such a withdrawal request by means of an if-then-else operator (exactly as in Example 2) and the predicate `m > n` testing whether or not the requested money exceeds the account's current balance. If this is the case, the request is rejected and the account goes into an overdraft state. Otherwise, the request is honored, the balance is updated, and the pending debt is decreased accordingly. What this rewrite rule clearly illustrates is that, although its lefthand side only involves constructors, its righthand side involves several defined functions needed to update the state, namely, the if-then-else operator, the `m > n` predicate, and the "monus" operator on natural numbers `_-_` used to decrease both the balance and the pending debt. Fortunately, the equations defining all these auxiliary functions are FVP, so that this rule, as well as the other rules in the example only involve Σ_1-terms. This means that this example meets the requirements for the input theory in the $\mathcal{R} \mapsto \overline{\mathcal{R}}^{\Omega}_{\Sigma_1,l,r}$ transformation. To give a flavor for the transformation itself, in which all the lefthand- and righthand-sides of the transformed rules become *constructor* terms, I list below the transformed rules for the above `[w]` rule. One feature of the terms below that might seem puzzling is the presence of the natural number addition operator `+`. The point is that `+` is a *free constructor modulo associativity-commutativity* axioms and the *identity* axiom for `0` (ACU), because the additive natural numbers are the free commutative monoid generated by `1`. As shown in [21], this yields a variant-based decision procedure for QF-satisfiability, not just for Presburger arithmetic, but for *all* other auxiliary functions, like monus and if-then-else, involved in this example.

```
rl [w] : < bal: n + m + x pend: m overdraft: false >
         # msgs,withdraw(m + x)
  =>
 < bal: n pend: 0 overdraft: false > # msgs .


rl [w] : < bal: n + m pend: m + x overdraft: false >
         # msgs,withdraw(m)
 =>
 < bal: n pend: x overdraft: false > # msgs .

rl [w] : < bal: n pend: y overdraft: false >
         # msgs',withdraw(1 + n + x)
 =>
 < bal: n pend: y overdraft: true > # msgs' .
```

The relevant question about this example is: *what is gained in translation?* And the relevant answer is: very much, particularly for narrowing-based reachability analysis. The reason is that, before the transformation, each narrowing step would take place by unifying a symbolic state with a rule's lefthand side *modulo* $E \uplus B$. Instead, now, the unification of symbolic states with lefthand sides of rules takes place modulo $B = B_{\Omega}$, that is, just modulo ACU, which is much more efficient that $E \uplus B$-unification by folding variant narrowing. In some sense, what has been achieved could be called a process of *total evaluation*, where the defined functions appearing in righthand sides of rules have been completely *evaluated away* by means of their constructor variants. Such total evaluation is what makes possible the reduction from $E \uplus B$-unification to just ACU-unification.

5 Related Work and Conclusions

Closely related work falls into three categories: (i) the already-mentioned symbolic reasoning techniques for rewrite theories, e.g., [1,2,10,12,16,17,23,25,26]; (ii) executability techniques for standard rewrite theories, including [8,28]; and (iii) variant-based symbolic computation, including [5,11,21,24], and also [7], where a limited form of "equational coherence completion" was introduced. In relation to all the work in (i)–(iii), the main contributions of this paper are: (1) a new notion of *generalized rewrite theory*, of rewriting in a generalized rewrite theory, and an initial model semantics for such theories; (2) new *symbolic executability requirements*, including a new notion of *coherence* that is a substantial generalization of the standard notions in [8,28]; and (3) new automatable *theory transformations* both to ensure ground coherence of generalized rewrite theories by *coherence completion*, and to make symbolic executability applicable to a widest possible class of such theories. It is worth noting that the new coherence completion transformations apply, in particular, to standard rewrite theories.

The most obvious next step is to implement all the theory transformations presented in Sect. 4. This can easily be done by computing variants in Maude,

and constructor variants in the Maude implementation of [24]. This will enable new applications, both in symbolic reasoning and in equational abstraction. It could also be used to substantially extend the features of the current Maude Coherence Checker [8].

Acknowledgments. I thank the referees for their constructive criticism and valuable suggestions to improve the paper. This work has been partially supported by NRL under contract number N00173-17-1-G002.

References

1. Arusoaie, A., Lucanu, D., Rusu, V.: Symbolic execution based on language transformation. Comput. Lang. Syst. Struct. **44**, 48–71 (2015)
2. Bae, K., Escobar, S., Meseguer, J.: Abstract logical model checking of infinite-state systems using narrowing. In: Rewriting Techniques and Applications (RTA 2013). LIPIcs, vol. 21, pp. 81–96. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2013)
3. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theor. Comput. Sci. **360**(1–3), 386–414 (2006)
4. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
5. Comon-Lundh, H., Delaune, S.: The finite variant property: how to get rid of some algebraic properties. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 294–307. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_22
6. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 243–320. North-Holland, Amsterdam (1990)
7. Durán, F., Lucas, S., Meseguer, J.: Termination modulo combinations of equational theories. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS (LNAI), vol. 5749, pp. 246–262. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04222-5_15
8. Durán, F., Meseguer, J.: On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. J. Algebr. Logic Program. **81**, 816–850 (2012)
9. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-69962-7
10. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: cryptographic protocol analysis modulo equational properties. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007–2009. LNCS, vol. 5705, pp. 1–50. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03829-7_1
11. Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. J. Algebr. Logic Program. **81**, 898–928 (2012)
12. Falke, S., Kapur, D.: Rewriting induction + Linear arithmetic = Decision procedure. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 241–255. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_20
13. Futatsugi, K.: Fostering proof scores in CafeOBJ. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 1–20. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16901-4_1

14. Goguen, J., Meseguer, J.: Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci. **105**, 217–273 (1992)
15. Jouannaud, J.P., Kirchner, H.: Completion of a set of rules modulo a set of equations. SIAM J. Comput. **15**, 1155–1194 (1986)
16. Kop, C., Nishida, N.: Automatic constrained rewriting induction towards verifying procedural programs. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 334–353. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12736-1_18
17. Lucanu, D., Rusu, V., Arusoaie, A., Nowak, D.: Verifying reachability-logic properties on rewriting-logic specifications. In: Martí-Oliet, N., Ölveczky, P.C., Talcott, C. (eds.) Logic, Rewriting, and Concurrency - Essays Dedicated to José Meseguer on the Occasion of His 65th Birthday. LNCS, vol. 9200, pp. 451–474. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23165-5_21
18. Lucas, S., Meseguer, J.: Normal forms and normal theories in conditional rewriting. J. Log. Algebr. Methods Program. **85**(1), 67–97 (2016)
19. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Presicce, F.P. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-64299-4_26
20. Meseguer, J.: Generalized rewrite theories and coherence completion. Technical report, University of Illinois Computer Science Department, March 2018. http://hdl.handle.net/2142/99546
21. Meseguer, J.: Variant-based satisfiability in initial algebras. Sci. Comput. Program. **154**, 3–41 (2018)
22. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theor. Comput. Sci. **403**(2–3), 239–264 (2008)
23. Rocha, C., Meseguer, J., Muñoz, C.A.: Rewriting modulo SMT and open system analysis. J. Log. Algebr. Methods Program. **86**, 269–297 (2017)
24. Skeirik, S., Meseguer, J.: Metalevel algorithms for variant satisfiability. J. Log. Algebr. Methods Program. **96**, 81–110 (2018)
25. Skeirik, S., Stefanescu, A., Meseguer, J.: A constructor-based reachability logic for rewrite theories. In: Fioravanti, F., Gallagher, J. (eds.) LOPSTR 2017. LNCS, vol. 10855, pp. 207–217. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94460-9_12. Technical report, University of Illinois Computer Science Department, March 2017. http://hdl.handle.net/2142/95770
26. Ştefănescu, A., Ciobâcă, Ş., Mereuta, R., Moore, B.M., Şerbănută, T.F., Roşu, G.: All-path reachability logic. In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 425–440. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_29
27. Thati, P., Meseguer, J.: Symbolic reachability analysis using narrowing and its application to the verification of cryptographic protocols. J. High. Order Symb. Comput. **20**(1–2), 123–160 (2007)
28. Viry, P.: Equational rules for rewriting logic. Theor. Comput. Sci. **285**, 487–517 (2002)

Proving Ground Confluence of Equational Specifications Modulo Axioms

Francisco Durán[1], José Meseguer[2], and Camilo Rocha[3](✉)

[1] Universidad de Málaga, Málaga, Spain
[2] University of Illinois, Urbana-Champaign, Champaign, IL, USA
[3] Pontificia Universidad Javeriana, Cali, Colombia
camilo.rocha@javerianacali.edu.co

Abstract. Terminating functional programs should be *deterministic*, i.e., should evaluate to a *unique* result, regardless of the evaluation order. For equational functional programs such determinism is exactly captured by the *ground confluence* property. For terminating equations this is equivalent to *ground local confluence*, which follows from *local confluence*. Checking local confluence by computing critical pairs is the *standard* way to check ground confluence. The problem is that some perfectly reasonable equational programs are *not* locally confluent and it can be very hard or even impossible to make them so by adding more equations. We propose a three-step strategy to prove that an equational program *as is* is ground confluent: *First*: apply the strategy proposed in [9] to use non-joinable critical pairs as *completion hints* to either achieve local confluence or reduce the number of critical pairs. *Second*: use the *inductive inference system* proposed in this paper to prove the remaining critical pairs ground joinable. *Third*: to show ground confluence of the original specification, prove also ground joinable the equations added. These methods apply to order-sorted and possibly conditional equational programs modulo axioms such as, e.g., Maude functional modules.

1 Introduction

Functional programs should be *deterministic*; that is, if they terminate for a given input, they should return a *unique* value, regardless of the evaluation order. *Ground confluence* is the precise characterization of such determinism for functional equational programs associated to equational theories of the form $\mathcal{E} = (\Sigma, E \uplus B)$, were B are structural axioms and E are, possibly conditional, equations that are executed as rewrite rules $\overrightarrow{E}$ *modulo* B. Therefore, for execution purposes, all the relevant information is contained in the rewrite theory $\mathcal{R}_{\mathcal{E}} = (\Sigma, B, \overrightarrow{E})$. Since ground confluence is essential both for correct execution and for almost any form of formal verification about properties of $\mathcal{E}$ and $\mathcal{R}_{\mathcal{E}}$, methods to prove ground confluence are very important.

The *standard method* to do so for a terminating equational program $\mathcal{R}_{\mathcal{E}} = (\Sigma, B, \overrightarrow{E})$ is to: (i) prove that it is indeed *operationally terminating* (and if Σ is

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 184–204, 2018.
https://doi.org/10.1007/978-3-319-99840-4_11

order-sorted, also *sort decreasing*); and then (ii) since operational termination plus local confluence imply confluence, prove the *stronger* property that $\mathcal{R}_{\mathcal{E}}$ is locally confluent (modulo B). This tends to work well in many cases, but not always. The thorny issue addressed in this paper is what to do when this standard method does not work.

In [9], the wild goose chase for a convergent specification by attempting a Knuth-Bendix completion of $\mathcal{E}$ was explicitly discouraged, since it can often lead to an infinite loop and, even if it were to succeed, can result in a highly bloated and hard to understand specification. Instead, the following *incremental strategy* in the spirit of Knuth-Bendix was suggested: since failure of a proof of local confluence will generate a set of unjoinable *critical pairs* characterizing the most general cases in which rules cannot be shown confluent, such critical pairs can be used as very useful *hints* for a user to try to either: (i) orient a critical pair as a rule and add it to the specification; or (ii) if the critical pair has the form $C[u] = C[v]$ with C a common context, orient instead $u = v$ and add it to the specification; or (iii) *generalize* $u = v$ in cases (i) and (ii) into a more general $u' = v'$ that has $u = v$ as a substitution instance and add an oriented version of $u' = v'$ to the specification. In this way, we obtain a new specification $\mathcal{R}_{\mathcal{E}'} = (\Sigma, B, \overrightarrow{E} \uplus \overrightarrow{G})$, where $\overrightarrow{G}$ are the new oriented equations added by methods (i)–(iii). If $\mathcal{R}_{\mathcal{E}'}$ is locally confluent, operationally terminating, and sort-decreasing, we are done; otherwise, we can *iterate* the process with the critical pairs obtained for $\mathcal{R}_{\mathcal{E}'}$.

In practice, this incremental strategy works reasonably well, but not always. Furthermore, it raises the following unsolved questions:

1. Have we *changed* the initial algebra semantics? That is, do the original $\mathcal{R}_{\mathcal{E}}$ and its extension $\mathcal{R}_{\mathcal{E}'}$ have the same initial algebra when viewed as equational theories? If only additions of type (i) are made, this is always true; but additions of type (ii)–(iii) are often needed in practice.
2. Was the *original* specification $\mathcal{R}_{\mathcal{E}}$ already ground confluent? That is, can we use $\mathcal{R}_{\mathcal{E}'}$ as the proverbial "Wittgenstein ladder" that we can kick away *after* we have proved its local confluence?
3. What do we do when we run into a *wall*? Specifically, the "wall" of having an equation $u = v$ obtained by methods (i)–(iii) above that *cannot* be oriented because it would lead to non-termination.

Our Contributions. This paper provides new methods that answer these three questions and can greatly help in proving an original specification ground confluent. In a nutshell, a more general and powerful strategy is proposed with three steps: *First*: use the above-described strategy from [9] as far as it can go. *Second*: if you hit the wall of non-orientability for some critical pairs (Question 3), prove the *ground joinability* of such remaining pairs by the *inductive methods* presented in this work. *Third*: to ensure preservation of the initial algebra semantics (Question 1) and the ground confluence of the original specification (Question 2), use the same inductive methods to prove ground joinability of all the equations added along the first step. Of course, one could skip the first step altogether and

merge the second and third steps into one; but this may require a considerably bigger effort, since the whole point of taking the first step is to *greatly reduce* the number of pairs to be proved ground joinable. Furthermore, the user may have made an *actual mistake* in the original specification $\mathcal{R}_\mathcal{E}$, so that the second and third steps become meaningless. In such a case, the first step can be quite helpful in identifying such mistakes and help the user *restart* the process with a new specification.

Paper Organization. Preliminaries are gathered in Sect. 2. The strategy's First Step is illustrated in Sect. 3 by a hereditarily finite sets specification that does indeed run into a non-orientability wall. The inductive inference system for ground confluence is presented and proved sound in Sect. 4, and is illustrated by proving the inductive joinability of the non-orientable critical pair from Sect. 3, thus illustrating the Second Step. The Third Step is then illustrated in detail for the running example in Sect. 5. Some related work and conclusions are discussed in Sect. 6. Results for the mechanized proofs and proofs for auxiliary results presented here can be found in [10].

2 Preliminaries

Notation on terms, term algebras, and equational theories is used as, e.g., in [12]. 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 typed with sorts in S. The binary relation $\equiv_\leq$ denotes the equivalence relation $(\leq \cup \geq)^+$ generated by $\leq$ on S and its point-wise extension to strings in S^*. The function symbols in F can be subsort-overloaded. For any sort $s \in S$, the expression $[s]$ denotes the connected component of s, that is, $[s] = [s]_{\equiv_\leq}$. A *top sort* in Σ is a sort $s \in S$ such that for all $s' \in [s]$, $s' \leq s$. Let $X = \{X_s\}_{s \in S}$ be an S-indexed family of disjoint variable sets with each X_s countably infinite. For any $s \in S$, let $X_{\leq s} = \bigcup_{s' \in S \wedge s' \leq s} X_{s'}$. The *set of terms of sort s* and the *set of ground terms of sort s* are denoted, respectively, by $T_\Sigma(X)_s$ and $T_{\Sigma,s}$; similarly, $T_\Sigma(X)$ and T_Σ denote, respectively, the set of terms and the set of ground terms. $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$ denote the corresponding order-sorted Σ-term algebras. All order-sorted signatures are assumed *preregular* [12], i.e., each Σ-term t has a unique *least sort* $ls(t) \in S$ s.t. $t \in T_\Sigma(X)_{ls(t)}$. It is also assumed that Σ has *nonempty sorts*, i.e., $T_{\Sigma,s} \neq \emptyset$ for each $s \in S$. The *set of variables* of t is written $vars(t)$ and for a list of terms $t_1, \ldots, t_n$, $vars(t_1, \ldots, t_n) = vars(t_1) \cup \cdots \cup vars(t_n)$.

A *substitution* is an S-indexed mapping $\theta \in [X \longrightarrow T_\Sigma(X)]$ that is different from the identity only for a finite subset of X and such that $\theta(x) \in T_\Sigma(X)_s$ if $x \in X_s$, for any $x \in X$ and $s \in S$. The expression $\theta|_Y$ denotes the restriction of θ to a family of variables $Y \subseteq X$. The *domain* of θ, denoted $dom(\theta)$, is the subfamily of X such that $x \in dom(\theta)$ iff $\theta(x) \neq x$, for each $x \in X$. If $dom(\theta) = \{x_1, \ldots, x_n\}$ we write $\theta = \{x_1 \mapsto \theta(x_1), \ldots, x_n \mapsto \theta(x_n)\}$. The *range* of θ is the set $ran(\theta) = \bigcup\{vars(\theta(x)) \mid x \in dom(\theta)\}$. 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 (in diagrammatic order) of two substitutions θ_1 and θ_2 is denoted by $\theta_1\theta_2$, so that $t\theta_1\theta_2$ denotes $(t\theta_1)\theta_2$. 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 $(t_1, \ldots, t_n) \mapsto C(t_1, \ldots, t_n) = c\theta$, where $\theta(x_i) = t_i$ for $1 \leq i \leq n$ and $\theta(x) = x$ for $x \notin \{x_1, \ldots, x_n\}$.

An *equational theory* is a tuple (Σ, E), with Σ an order-sorted signature and E a finite collection of (possibly conditional) Σ-equations. 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 $\mathcal{E} \vdash t = u$, where $\mathcal{E} \vdash t = u$ denotes $\mathcal{E}$-provability by the deduction rules for order-sorted equational logic in [14]. For the purpose of this paper, such inference rules, which are analogous to those of many-sorted equational logic, are even simpler thanks to the assumption that Σ has nonempty sorts, which makes unnecessary the explicit treatment of universal quantifiers. The expressions $\mathcal{T}_\mathcal{E}(X)$ 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).

We assume acquaintance with the usual notions of position p in a term t, subterm $t|_p$ at position p, and term replacement $t[u]_p$ at position p (see, e.g., [5]). 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 possibly conditional Σ-rules, with conditions being a conjunction of Σ-equalities. A rewrite theory $\mathcal{R}$ 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 $(l \rightarrow r \textit{ if } \phi) \in R$, a term t', a position p in t', and a substitution $\theta : X \longrightarrow T_\Sigma(X)$ satisfying $t =_E t' = t'[l\theta]_p$, $u =_E t'[r\theta]_p$, and $(\Sigma, E) \vdash \phi\theta$. The tuple $\mathcal{T}_\mathcal{R} = (\mathcal{T}_{\Sigma/E}, \rightarrow^\star_\mathcal{R})$, where, by definition, $\rightarrow^\star_\mathcal{R} = \rightarrow^+_\mathcal{R} \cup =_E$, where $\rightarrow^+_\mathcal{R}$ denotes the transitive closure of $\rightarrow_\mathcal{R}$, is called the *initial reachability model of* $\mathcal{R}$ [3].

In this paper we will mostly focus on rewrite theories of the form $\mathcal{R}_\mathcal{E} = (\Sigma, B, \overrightarrow{E})$ associated to an equational theory $\mathcal{E} = (\Sigma, E \uplus B)$, were: (i) B are decidable structural axioms whose equations $u = v \in B$ are linear (no repeated variables in either u or v) and regular (same variables in u and v), for which a matching algorithm exists, and (ii) the possibly conditional rewrite rules $\overrightarrow{E}$ are *strictly* B*-coherent* [15]. Under such assumptions, the rewrite relation $t \rightarrow_{\mathcal{R}_\mathcal{E}} u$ holds iff there exists u' such that $u' =_B u$, and $t \rightarrow_{\overrightarrow{E},B} u'$, where, by definition, $t \rightarrow_{\overrightarrow{E},B} u'$ iff there exists a rule $(l \rightarrow r \textit{ if } \phi) \in \overrightarrow{E}$, a position p in t and a substitution θ such that $t|_p =_B l\theta$, $u' = t[r\theta]_p$, and $\mathcal{R}_\mathcal{E} \vdash \phi\theta$. We will assume throughout that the rules $\overrightarrow{E}$ are *always strictly* B-coherent. We finally assume that the axioms B are: (i) *sort-preserving*, i.e., for each $(u = v) \in B$ and substitution σ we have $ls(u\sigma) = ls(v\sigma)$; and (ii) *term-size preserving*[1], i.e., if $t =_B t'$, then $|t| = |t'|$.

[1] For combinations of associativity, commutativity, and identity axioms, this last condition only rules out identity axioms. However, both for termination and confluence analysis purposes, identity axioms can always be turned into convergent rewrite rules modulo associativity and/or commutativity axioms, as explained in [8].

Appropriate requirements are needed to make an equational theory $\mathcal{E} = (\Sigma, E \uplus B)$ *admissible* as an equational program, i.e., for making $\mathcal{R}_\mathcal{E} = (\Sigma, B, \overrightarrow{E})$ *executable* in languages such as Maude [4]. In this paper, besides the above assumptions about B and $\overrightarrow{E}$, we assume that the rules in $\overrightarrow{E}$ are sort-decreasing, operationally terminating, and ground confluent modulo B. The rewrite rules $\overrightarrow{E}$ are *sort decreasing* modulo B iff for each $(t \to u \textit{ if } \gamma) \in \overrightarrow{E}$ and substitution θ, $\mathit{ls}(t\theta) \geq \mathit{ls}(u\theta)$ if $\mathcal{R}_\mathcal{E} \vdash \gamma\theta$. $\mathcal{R}_\mathcal{E} = (\Sigma, B, \overrightarrow{E})$ is *operationally terminating* modulo B [6] iff there is no infinite well-formed proof tree in $(\Sigma, B, \overrightarrow{E})$. Call $t, t' \in T_\Sigma(X)$ *joinable* in $\mathcal{R}_\mathcal{E}$, denoted $t \downarrow_{\mathcal{R}_\mathcal{E}} t'$ iff there exist u, v such that $t \to^*_{\overrightarrow{E},B} u$, $t' \to^*_{\overrightarrow{E},B} v$, and $u =_B v$. Call $\mathcal{R}_\mathcal{E} = (\Sigma, B, \overrightarrow{E})$ *confluent* (resp., *ground confluent*) modulo B iff for all $t, t_1, t_2 \in T_\Sigma(X)$ (resp., for all $t, t_1, t_2 \in T_\Sigma$), if $t \to^*_{\overrightarrow{E},B} t_1$ and $t \to^*_{\overrightarrow{E},B} t_2$, then $t_1 \downarrow_{\mathcal{R}_\mathcal{E}} t_2$. For $\mathcal{R}_\mathcal{E} = (\Sigma, B, \overrightarrow{E})$ to have good executability properties as a terminating equational program, the following requirements are needed: (a) sort decreasingness, (b) operational termination, and (c) ground confluence. If conditions (a)–(c) are met, we call $\mathcal{R}_\mathcal{E}$ *ground convergent*. $\mathcal{R}_\mathcal{E}$ is called *convergent* if it satisfies the stronger requirements of sort-decreasingness, operational termination, and confluence.

3 An Equational Specification for Hereditarily Finite Sets

When checking the confluence of an equational specification, the CRC tool [9,11] provides as result a set of critical pairs that cannot be joined automatically by its built-in heuristics. They are proof obligations that can either be proved joinable or used as guidance for modifying the input specification. The methodology proposed in [9] for using the CRC tool suggests that critical pairs can help in identifying theorems of the original specification which, when added to it, may lead to a confluent or ground confluent specification. However, as the example of HF-SETS presented in this section shows, the analysis of critical pairs to modify a specification, though a useful first strategy, may be insufficient to make the specification ground confluent. Other techniques, such as the ones presented in Sect. 4, may be needed.

Consider the specification of hereditarily finite sets in Fig. 1, namely, of finite sets whose elements are all hereditarily finite sets (see, e.g., [13]). The recursive definition of well-founded hereditary sets has the empty set as the base case and if $s_1, \ldots, s_k$ are hereditarily finite, then so is $\{s_1, \ldots, s_k\}$. These sets play a key role in axiomatic set theory because they are a model of all the axioms of set theory except for the axiom of infinity. Furthermore, as the methods developed in this work will show, the initial model of the HF-SETS specification below is a *consistent* model of set theory without the axiom of infinity.

The Church-Rosser check of the HF-SETS module using the CRC tool says that the specification is sort-decreasing, but it cannot show that it is locally confluent, returning eight critical pairs as proof obligations. At this point, there are two alternatives: either (i) we try to prove the ground joinability of these critical

```
fmod HF-SETS is
  protecting BOOL-OPS .
  sorts Magma Set .
  subsort Set < Magma .
  op _,_ : Magma Magma -> Magma [ctor assoc comm] .
  op {_} : Magma -> Set [ctor] .
  op {} : -> Set [ctor] .

  vars M M' : Magma .
  vars S S' : Set .

  eq [01]: M, M = M .

  op _in_ : Magma Set -> Bool .                    *** set membership
  eq [11]: S in {S} = true .
  eq [12]: S in {} = false .
  eq [13]: {} in {{M}} = false .
  eq [14]: {M} in {{}} = false .
  eq [15]: {M} in {{M'}} = M in {M'} and M' in {M} .
  eq [16]: S in {S', M} = S in {S'} or S in {M} .
  eq [17]: (S, M) in S' = (S in S') and (M in S') .

  op _~_ : Set Set -> Bool .                       *** set equality
  eq [21]: S ~ S' = (S <= S') and (S' <= S) .

  op _<=_ : Set Set -> Bool .                      *** set containment
  eq [31]: {} <= S = true .
  eq [32]: {M} <= S = M in S .

  op _U_ : Set Set -> Set [assoc comm] .           *** union
  eq [41]: S U {} = S .
  eq [42]: {M} U {M'} = {M, M'} .
  eq [43]: S U {M} U {M'} = S U {M, M'} .

  op P : Set -> Set .                             *** powerset
  eq [51]: P({}) = {{}} .
  eq [52]: P({S}) = {{},{S}} .
  eq [53]: P({S, M}) = P({M}) U augment(P({M}), S) .

  op augment : Set Set -> Set .                   *** augmentation
  eq [61]: augment({}, S) = {} .
  eq [62]: augment({S}, S') = {{S'} U S} .
  eq [63]: augment({M, M'}, S) = augment({M}, S) U augment({M'}, S) .

  op _&_ : Set Set -> Set .                       *** intersection
  eq [71]: {} & S = {} .
 ceq [72]: {S} & S' = {S} if S in S' = true .
 ceq [73]: {S} & S' = {} if S in S' = false .
 ceq [74]: {S, M} & S' = {S} U ({M} & S') if S in S' = true .
 ceq [75]: {S, M} & S' = {M} & S' if S in S' = false .

  op <_;_> : Set Set -> Set .                     *** ordered pairs
  op <_;_> : Magma Magma -> Magma .               *** extension to magmas
  eq [91]: < S ; S' > = {S, {S, S'}} .
  eq [92]: < S ; S', M > = {S, {S, S'}}, < S ; M > .
  eq [93]: < S, M ; M' > = < S ; M' >, < M ; M' > .

  op _X_ : Set Set -> Set .                       *** cartesian product
  eq [101]: {} X S = {} .
  eq [102]: S X {} = {} .
  eq [103]: {M} X {M'} = {< M ; M' >} .
endfm
```

Fig. 1. Equational specification of hereditarily finite sets in Maude

pairs to conclude that the specification is locally ground confluent, or (ii) we follow the iterative strategy proposed in [9] to get a locally confluent specification or at least reduce the number of critical pairs for which a proof of joinability is necessary. In the rest of this section, we explore the second alternative. The first alternative will be revisited after the second one is exhausted (both are useful) in Sect. 5.

The following one is one of the critical pairs returned by the check:

```
cp HF-SETS1123 for 11 and 15
  true = M':Magma in {M':Magma} .
```

It comes from the overlap of equations 11 and 15. Although there are equations for all possible instances of the term `M in {M}`, Maude cannot reduce it as magmas. We can attempt adding equations to reduce it as follows:

```
fmod HF-SETS-0 is
  protecting HF-SETS .
  vars M M' : Magma .
  eq [18]: M in {M} = true .
  eq [19]: M in {M', M} = true .
endfm
```

A check of the Church-Rosser property for HF-SETS-0 returns seven critical pairs. Let us consider one of these critical pairs:

```
cp HF-SETS-095 for 01 and 63
  augment({M':Magma}, S:Set) = augment({M':Magma}, S:Set) U augment({M':Magma}, S:Set) .
```

This critical pair comes from the overlap of equations 01 and 63. Indeed, this critical pair cannot be further reduced because there is no idempotency equation for the union operator on sets. We can see the same problem in other four of the critical pairs reported by the tool. Although S U S = S could be proven in HF-SETS-0, there is the alternative option of extending the specification with an idempotency equation for set union.

```
fmod HF-SETS-1 is
  protecting HF-SETS-0 .
  var S : Set .
  eq [44]: S U S = S .
endfm
```

The Church-Rosser checker tool produces the following output for HF-SETS-1:

```
The following critical pairs must be proved joinable:
  cp HF-SETS-118 for 53 and 53
     P({#6:Magma}) U augment(P({#6:Magma}), S:Set) U augment(P({#6:Magma}) U
     augment(P({#6:Magma}), S:Set), #1:Set)
   = P({#6:Magma}) U augment(P({#6:Magma}), #1:Set) U augment(P({#6:Magma}) U
     augment(P({#6:Magma}), #1:Set), S:Set).
  cp HF-SETS-1355 for 01 and 53
     P({#3:Magma}) U augment(P({#3:Magma}), S:Set)
   = P({#3:Magma}) U augment(P({#3:Magma}), S:Set) U augment(P({#3:Magma}) U
     augment(P({#3:Magma}), S:Set), S:Set).
  The module is sort-decreasing.
```

A careful study of these critical pairs suggests the need for an equation to apply augment over the union operator.

```
fmod HF-SETS-2 is
  protecting HF-SETS-1 .
  vars S S' T : Set .
  eq [64]: augment(S U S', T) = augment(S, T) U augment(S', T) .
endfm
```

The number of critical pairs gets further decreased in HF-SETS-2, but two remain:

```
The following critical pairs must be proved joinable:
  cp HF-SETS-218 for 53 and 53
     P({#6:Magma}) U augment(P({#6:Magma}), S:Set) U augment(P({#6:Magma}), #1:Set) U
     augment(augment(P({#6:Magma}), S:Set), #1:Set)
   = P({#6:Magma}) U augment(P({#6:Magma}), S:Set) U augment(P({#6:Magma}), #1:Set) U
     augment(augment(P({#6:Magma}), #1:Set), S:Set).
  cp HF-SETS-2411 for 01 and 53
     P({#3:Magma}) U augment(P({#3:Magma}), S:Set)
   = P({#3:Magma}) U augment(P({#3:Magma}), S:Set)
                   U augment(augment(P({#3:Magma}), S:Set), S:Set).
  The module is sort-decreasing.
```

The second critical pair suggests the need for an equation handling the repeated application of the augment operator.

```
fmod HF-SETS-3 is
  protecting HF-SETS-2 .
  vars S T : Set .
  eq [65]: augment(augment(S, T), T) = augment(S, T) .
endfm
```

However, one critical pair remains in HF-SETS-3:

```
Church-Rosser check for HF-SETS-3
The following critical pairs must be proved joinable:
  cp HF-SETS-318 for 53 and 53
     P({#6:Magma})U augment(P({#6:Magma}),S:Set)U augment(P({#6:Magma}),#1:Set)U
     augment(augment(P({#6:Magma}),S:Set),#1:Set)
   = P({#6:Magma})U augment(P({#6:Magma}),S:Set)U augment(P({#6:Magma}),#1:Set)U
     augment(augment(P({#6:Magma}),#1:Set),S:Set).
  The module is sort-decreasing.
```

It is not obvious at all how to eliminate this critical pair, since adding the equation

```
 eq augment(augment(S, S'), T) = augment(augment(S, T), S') .
```

would make the specification *non-terminating*. This suggests that the second approach, i.e., the strategy of trying to complete the specification by analyzing the unjoinable critical pairs has now been exhausted. However, the original problem has now been reduced to a *single* critical pair. At this point, the best approach is to prove the *inductive joinability* of the critical pair HF-SETS-318 obtained in the check of HF-SETS-3, and thus conclude that the specification is ground locally confluent. Section 4 presents techniques for carrying out such inductive proofs. Indeed, it will also present results showing that the *original specification was already ground confluent!*, without the need for the extra equations added in the process. The specification is terminating. Indeed, the MTT tool [7,11] is able to find termination proofs for all the versions of the HF-SETS

module, and specifically for `HF-SETS-3` (see [10, Appendix B]). A proof of the sufficient completeness of the specification can be found in [10, Appendix C].

Finally, note that if an added equation comes from orienting a critical pair, it is a logical consequence of the specification and therefore the new specification has the *same* initial model of the old one. Although the additional equations added during the process may not be those obtained from critical pairs as such, proving that they are ground joinable is enough to show that they are actually *inductive lemmas*, and therefore—as explained in more detail in Theorem 6 in Sect. 4—that they both preserve the initial algebra semantics *and* can be *removed* from the original specification.

4 Proving Ground Joinability

This section presents inductive techniques for proving ground joinability for rewrite theories associated to equational specifications. These techniques are presented as meta-theorems about the ground reachability relation induced by a rewrite theory and are used to justify the inference system also presented in this section.

Definition 1. *Let $\mathcal{R}$ be a rewrite theory with signature $\Sigma = (S, \leq, F)$ and $t, u \in T_\Sigma(X)_s$ for some $s \in S$. The terms t and u are called:*

1. $\mathcal{R}$-joinable, *written $\mathcal{R} \vdash (\forall X)\, t \downarrow u$, iff there is $v \in T_\Sigma(X)_s$ such that $\mathcal{R} \vdash (\forall X)\, t \rightarrow^\star v$ and $\mathcal{R} \vdash (\forall X)\, u \rightarrow^\star v$.*
2. ground $\mathcal{R}$-joinable, *written $\mathcal{R} \Vdash (\forall X)\, t \downarrow u$, iff $\mathcal{R} \vdash t\theta \downarrow u\theta$ for all ground substitutions $\theta \in [X \longrightarrow T_\Sigma]$.*

The authors of [18] investigate constructor-based inductive techniques for proving ground joinability. They distinguish two notions of constructors for a rewrite theory $\mathcal{R}$, namely, one for the equations and another one for the rules in $\mathcal{R}$.

Definition 2 (Definitions 5 and 6 [18]). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory with underlying equational theory $\mathcal{E} = (\Sigma, E)$. A* constructor signature pair *for $\mathcal{R}$ is a pair (Υ, Ω) of order-sorted subsignatures $\Upsilon = (S, \leq, F_\Upsilon) \subseteq \Omega = (S, \leq, F_\Omega)$. The sets of terms $T_\Upsilon = \{T_{\Upsilon,s}\}_{s\in S}$ and $T_\Omega = \{T_{\Omega,s}\}_{s\in S}$ are called, respectively,* E-constructor terms *and* $\mathcal{R}$-constructor terms. *The rewrite theory $\mathcal{R}$ is called:*

1. E-sufficiently complete relative to Ω *iff $(\forall s \in S)(\forall t \in T_{\Sigma,s})(\exists u \in T_{\Omega,s})\; \mathcal{E} \vdash t = u$.*
2. $\mathcal{R}$-sufficiently complete relative to Υ *iff $(\forall s \in S)(\forall t \in T_{\Sigma,s})(\exists v \in T_{\Upsilon,s})\; \mathcal{R} \vdash t \rightarrow^\star v$.*
3. sufficiently complete relative to (Υ, Ω) *iff (1) and (2) hold.*

The notion of sufficient completeness for a rewrite theory $\mathcal{R}$ relative to a constructor signature pair (Υ, Ω) is that $\Omega \subseteq \Sigma$ are the constructors for the equations and $\Upsilon \subseteq \Omega$ the constructors for the rules, thus including the standard concept of constructor for equational specifications as a special case. The intuition behind equational constructor terms is that any ground Σ-term should be *provably equal* to a term in T_Ω and for rewrite constructors that any Σ-term should be *rewritable* to a term in T_Υ.

It is sufficient to consider all $\mathcal{R}$-constructor terms in $T_{\Upsilon,s}$ when inducting on a variable x of sort s, for a proof on inductive joinability in $\mathcal{R}$ to be sound.

Theorem 1 (Theorem 6 [18]). *Let $\mathcal{R}$ be a rewrite theory with signature $\Sigma = (S, \leq, F)$ and $t, u \in T_\Sigma(X)_s$ for some $s \in S$. If $\mathcal{R}$ is sufficiently complete relative to the constructor signature pair (Υ, Ω), then $\mathcal{R} \Vdash (\forall X)\, t \downarrow u$ iff $(\forall \eta \in [X \longrightarrow T_\Upsilon])\ \mathcal{R} \vdash t\eta \downarrow u\eta$.*

Figure 2 presents the JOIN, CTX and GRAL inference rules for proving joinability for a rewrite theory $\mathcal{R}$, respectively, by rewrite-based reasoning, inductive reasoning under contexts, and generalization. The soundness of the JOIN rule is straightforward to obtain, while Theorem 2 justifies the soundness of the CTX and GRAL rules. This result can be used to simplify the complexity of terms to be joinable if they share a common context.

$$\frac{\mathcal{R} \vdash (\forall X)\, t \downarrow u}{\mathcal{R} \Vdash (\forall X)\, t \downarrow u}\ \textsc{Join} \qquad \frac{\mathcal{R} \Vdash (\forall X)\, t \downarrow u}{\mathcal{R} \Vdash (\forall X)\, C[t] \downarrow C[u]}\ \textsc{Ctx} \qquad \frac{\mathcal{R} \Vdash (\forall X)\, t \downarrow u}{\mathcal{R} \Vdash (\forall X)\, t\theta \downarrow u\theta}\ \textsc{Gral}$$

Fig. 2. Inference rules for proving joinability for a rewrite theory $\mathcal{R}$ by rewrite-based reasoning, and inductive reasoning for contexts and substitution instances.

Theorem 2. *Let $\mathcal{R}$ be a rewrite theory with signature $\Sigma = (S, \leq, F)$ and $C[t], C[u] \in T_\Sigma(X)_s$ for some $s \in S$. If $\mathcal{R} \Vdash (\forall X)\, t \downarrow u$, then:*

1. *$\mathcal{R} \Vdash (\forall X)\, C[t] \downarrow C[u]$;*
2. *$\mathcal{R} \Vdash (\forall X)\, t\theta \downarrow u\theta$, for any substitution $\theta \in [X \longrightarrow T_\Sigma(X)]$.*

Proof. The two properties follow from the fact that the rewrite relation $\rightarrow_\mathcal{R}$ is closed under contexts and substitutions. □

Since the goal is to prove ground joinability of a rewrite theory of the form $\mathcal{R}_\mathcal{E} = (\Sigma, B, \overrightarrow{E})$ associated to an equational theory $\mathcal{E} = (\Sigma, E \uplus B)$, such as that for hereditarily finite sets presented in Sect. 3, the most appropriate notion of constructor is that of $\mathcal{R}_\mathcal{E}$-constructors. More precisely, a constructor signature pair for $\mathcal{R}_\mathcal{E}$ has always the form (Υ, Σ) because the only equations in $\mathcal{R}_\mathcal{E}$ are the axioms B not associated to any rewriting. Hence, $\mathcal{R}_\mathcal{E}$ sufficient completeness is *always* relative only to Υ. One more remark is important for what follows. As pointed out in Sect. 2, we assume that $\mathcal{R}_\mathcal{E} = (\Sigma, B, \overrightarrow{E})$ is admissible (except for

its ground confluence, which may remain to be proved). In particular this means that $\mathcal{R}_\mathcal{E}$ is strictly B-coherent in the sense of [15]. Therefore, the two notions of joinability (resp. ground joinability) involved, namely the one in Definition 1, and that defined in terms of the rewrite relation $\rightarrow_{\overrightarrow{E},B}$ in Sect. 2 actually coincide (see [15]). We will implicitly use this agreement between both notions in what follows.

Reasoning about ground joinability requires inductive inference support, e.g., in the form of a constructor-based scheme using finite generating sets.

Definition 3. *Let $\mathcal{E} = (\Sigma, E \uplus B)$ be an equational theory, with $\Sigma = (S, \leq, F)$, such that the rewrite theory $\mathcal{R}_\mathcal{E}$ is weakly terminating, ground sort-decreasing, and has subsignature Υ of $\mathcal{R}_\mathcal{E}$-constructors. Further, let $s \in S$. A set $G_s \subseteq T_{\Upsilon,s}(X)$ is a* (finite) generating set for s modulo B *iff G_s is finite, $G_s \cap X = \emptyset$, and*

$$T_{\Upsilon/B,s} = \bigcup_{w \in G_s} \{[w\sigma]_B \mid \sigma \in [vars(w) \longrightarrow T_\Upsilon]\}\,.$$

The following induction scheme is sound for inferring ground joinability in $\mathcal{R}_\mathcal{E}$.

Theorem 3. *Let $\mathcal{R}_\mathcal{E}$ be a weakly terminating and ground sort-decreasing rewrite theory, with signature $\Sigma = (S, \leq, F)$ and subsignature Υ of $\mathcal{R}_\mathcal{E}$-constructors. Moreover, let $t, u \in T_\Sigma(X)$, $x \in vars(t, u) \cap X_s$ for some $s \in S$, and G_s a generating set for s modulo B, such that (without loss of generality) $vars(G_s) \cap vars(t, u) = \emptyset$. Then:*

$$\textit{If } \mathcal{R}_\mathcal{E} \Vdash (\forall X) \bigwedge_{w \in G_s} \left[\bigwedge_{y \in vars(w) \cap X_{\leq s}} (t \downarrow u)\{x \mapsto y\} \right] \Rightarrow (t \downarrow u)\{x \mapsto w\},$$

then $\mathcal{R}_\mathcal{E} \Vdash (\forall X)\, t \downarrow u$.

Proof. By contradiction. Suppose the antecedent holds, but there is a ground substitution $\sigma \in [vars(t, u) \longrightarrow T_\Sigma]$ such that $\mathcal{R}_\mathcal{E} \not\vdash (t \downarrow u)\sigma$. Note, however, that by $\overrightarrow{E}$ being strict B-coherent and G_s being a generating set for s modulo B, σ is always of the form $\sigma =_B \{x \mapsto w\}\tau$, for some $w \in G_s$ and substitution τ, and then we have

$$\mathcal{R}_\mathcal{E} \not\vdash (t \downarrow u)\sigma \quad \text{iff} \quad \mathcal{R}_\mathcal{E} \not\vdash (t \downarrow u)\{x \mapsto w\}\tau.$$

Consider now the non-empty set of ground terms

$$\{w\tau \mid w \in G_s \;\wedge\; \tau \in [Y_w \longrightarrow T_\Sigma] \;\wedge\; \mathcal{R}_\mathcal{E} \not\vdash (t \downarrow u)\{x \mapsto w\}\tau\}$$

where $Y_w = (vars(t, u) \setminus \{x\}) \cup vars(w)$. Pick $w\tau_0$ of smallest term size possible in the above set. By the strict B-coherence of $\overrightarrow{E}$ and the assumption that the axioms B are size-preserving, this means that for any ground substitution

$\sigma \in [vars(t,u) \longrightarrow T_\Sigma]$, such that $\mathcal{R}_\mathcal{E} \nvdash (t \downarrow u)\sigma$, we must have $|\sigma(x)| \geq |w\tau_0|$. In particular, since $w \cap X = \emptyset$, this means that for each $y \in vars(w) \cap X_{\leq s}$ we must have $|\tau_0(y)| < |w\tau_0|$ and therefore $\mathcal{R}_\mathcal{E} \vdash (t \downarrow u)\{x \mapsto y\}\tau_0$. But, by hypothesis this implies $\mathcal{R}_\mathcal{E} \vdash (t \downarrow u)\{x \mapsto w\}\tau_0$, a contradiction. □

It is also sound to reason about ground joinability in $\mathcal{R}_\mathcal{E}$ using case analysis based on the $\mathcal{R}_\mathcal{E}$-constructor signature Υ (Fig. 3).

$$\frac{\mathcal{R}_\mathcal{E} \Vdash (\forall X) \bigwedge_{w \in G_s} \left[\bigwedge_{y \in vars(w) \cap X_{\leq s}} (t \downarrow u)\{x \mapsto y\} \right] \Rightarrow (t \downarrow u)\{x \mapsto w\}}{\mathcal{R}_\mathcal{E} \Vdash (\forall X)\, t \downarrow u} \textsc{GSInd}$$

$$\frac{\mathcal{R}_\mathcal{E} \Vdash (\forall X) \bigwedge_{w \in G_s} (t \downarrow u)\{x \mapsto w\}}{\mathcal{R}_\mathcal{E} \Vdash (\forall X)\, t \downarrow u} \textsc{CtorCases}$$

Fig. 3. Inference rules for proving ground joinability for a rewrite theory $\mathcal{R}_\mathcal{E}$ with $\mathcal{R}_\mathcal{E}$-constructors Υ by induction relative to the generating set G_s and by constructor-based case analysis on a variable $x \in vars(t,u) \cap X_s$.

Theorem 4. *Let $\mathcal{R}_\mathcal{E}$ be a weakly terminating and ground sort-decreasing rewrite theory, with signature $\Sigma = (S, \leq, F)$ and subsignature Υ of $\mathcal{R}_\mathcal{E}$-constructors. Moreover, let $t, u \in T_\Sigma(X)$, $x \in vars(t,u) \cap X_s$ for some $s \in S$, and G_s a generating set for s modulo B, such that (without loss of generality) $vars(G_s) \cap vars(t,u) = \emptyset$. Then:*

$$\mathcal{R}_\mathcal{E} \Vdash (\forall X)\, t \downarrow u \quad \textit{iff} \quad \mathcal{R}_\mathcal{E} \Vdash (\forall X) \bigwedge_{w \in G_s} (t \downarrow u)\{x \mapsto w\}.$$

Proof. If $\mathcal{R}_\mathcal{E} \Vdash (\forall X)\, t \downarrow u$, then clearly $\mathcal{R}_\mathcal{E} \Vdash (\forall X) \bigwedge_{w \in G_s} (t \downarrow u)\{x \mapsto w\}$. For the proof in the opposite direction, let $\sigma \in [X \longrightarrow T_\Sigma]$ be such that $\mathcal{R}_\mathcal{E} \nvdash (t \downarrow u)\sigma$: the goal is to show that $\mathcal{R}_\mathcal{E} \nVdash (\forall X) \bigwedge_{w \in G_s} (t \downarrow u)\{x \mapsto w\}$, for some $w \in G_s$. Since G_s is a generating set for the sort s and $x \in X_s$, then there is $w \in G_s$ and $\rho \in [X \longrightarrow T_\Sigma]$ such that $\sigma(x) =_B w\rho$. Let $\sigma' = \sigma_{|vars(t,u) \setminus \{x\}} \uplus \rho$ and observe that σ' is well-defined because of the assumption $vars(G_s) \cap vars(t,u) = \emptyset$. Furthermore, observe:

$$\begin{aligned} (t \downarrow u)\sigma &= (t \downarrow u)\{x \mapsto \sigma(x)\}\sigma_{|vars(t,u) \setminus \{x\}} \\ &=_B (t \downarrow u)\{x \mapsto w\rho\}\sigma_{|vars(t,u) \setminus \{x\}} \\ &= (t \downarrow u)\{x \mapsto w\}(\sigma_{|vars(t,u) \setminus \{x\}} \uplus \rho) \\ &= (t \downarrow u)\{x \mapsto w\}\sigma'. \end{aligned}$$

Hence, by the strict B-coherence of $\overrightarrow{E}$, we must have $\mathcal{R}_\mathcal{E} \nVdash (\forall X) \bigwedge_{w \in G_s} (t \downarrow u)\{x \mapsto w\}$. □

This concludes the inference system for proving ground joinability. However, an important practical issue remains: how should the checking of $\mathcal{R} \vdash (\forall X)\, t \downarrow u$ used in inference rule Join be best *mechanized*? After all, $t \downarrow u$ is a somewhat *complex* relation, involving existential quantification. This issue can be satisfactorily addressed by means of a program transformation $\mathcal{R}_{\mathcal{E}} \mapsto \mathcal{R}_{\mathcal{E}}^{\approx}$ that extends the possibly conditional and operationally terminating rewrite theory $\mathcal{R}_{\mathcal{E}}$, associated to an equational theory $\mathcal{E} = (\Sigma, E \uplus B)$, to a theory $\mathcal{R}_{\mathcal{E}}^{\approx}$ with: (i) a new sort *Prop* with constant `tt` and (ii) a new operator $_ \approx _$ with the rule $x \approx x \to tt$, such that

$$\mathcal{R}_{\mathcal{E}} \vdash (\forall X)\, t \downarrow u \quad \text{iff} \quad \mathcal{R}_{\mathcal{E}}^{\approx} \vdash (\forall X)\, t \approx u \to^{\star} tt.$$

Since the right side of the equivalence is a *reachability property* and the transformation $\mathcal{R}_{\mathcal{E}} \mapsto \mathcal{R}_{\mathcal{E}}^{\approx}$ preserves operational termination, the theory $\mathcal{R}_{\mathcal{E}}^{\approx}$ and Maude's *search* command can be used to check that $\mathcal{R} \vdash (\forall X)\, t \downarrow u$. This is used in the Example 1 below, where the binary function symbol `join` implements the operator $_ \approx _$. The precise description of the $\mathcal{R}_{\mathcal{E}} \mapsto \mathcal{R}_{\mathcal{E}}^{\approx}$ transformation is given in [10, Appendix D].

Example 1. Recall from Sect. 3 the only critical pair output by the CRC tool for the HF-SETS-3 specification; the goal is to prove:

$$\textsf{HF-SETS-3} \Vdash (\forall M : Magma; S, T : Set)\, t(M, S, T) \downarrow u(M, S, T)$$

where

$$\begin{aligned} t(M,S,T) = P(\{M\}) \cup\ & augment(P(\{M\}), S) \cup augment(P(\{M\}), T) \\ \cup\ & augment(augment(P(\{M\}), S), T) \\ u(M,S,T) = P(\{M\}) \cup\ & augment(P(\{M\}), S) \cup augment(P(\{M\}), T) \\ \cup\ & augment(augment(P(\{M\}), T), S) \end{aligned}$$

By the Ctx rule it suffices to prove:

$$\begin{aligned} &\textsf{HF-SETS-3} \Vdash (\forall M : Magma; S, T : Set) \\ &\qquad augment(augment(P(\{M\}), S), T) \downarrow augment(augment(P(\{M\}), T), S) \end{aligned}$$

Moreover, since $P(\{M\})$ has sort Set, this statement can be proved by considering a stronger property, namely, by using the Gral rule and proving:

$$\textsf{HF-SETS-3} \Vdash (\forall S, S', T : Set)\ augment(augment(S', S), T) \downarrow augment(augment(S', T), S)$$

This proof obligation can be dealt with by using the CTORCASES rule on $S' \in X_{\mathsf{Set}}$ with generating set $G_{\mathsf{Set}} = \{\{\}, \{M\}\}$ and $M \in X_{\mathsf{Magma}}$. This rule application results in the following two proof obligations:

$$\mathsf{HF\text{-}SETS\text{-}3} \Vdash (\forall S, T : Set)\ augment(augment(\{\}, S), T) \downarrow augment(augment(\{\}, T), S)$$
$$\mathsf{HF\text{-}SETS\text{-}3} \Vdash (\forall S, T : Set; M : Magma)$$
$$augment(augment(\{M\}, S), T) \downarrow augment(augment(\{M\}, T), S)$$

The first proof obligation can be discharged by a search command in $\mathcal{R}^{\approx}_{\mathsf{HF\text{-}SETS\text{-}3}}$:

```
search in HF-SETS-3-REACH :
  join(augment(augment({{}}, S), T), augment(augment({{}}, T), S)) =>! tt .
Solution 1 (state 1)
```

The second proof obligation can be handled using the GSIND rule on $M \in X_{\mathsf{Magma}}$ with generating set $G_{\mathsf{Magma}} = \{S', (S', M')\}$, $S' \in X_{\mathsf{Set}}$, and $M' \in X_{\mathsf{Magma}}$:

$$\mathsf{HF\text{-}SETS\text{-}3} \vdash (\forall S, S', T : Set)$$
$$augment(augment(\{S'\}, S), T) \downarrow augment(augment(\{S'\}, T), S)$$
$$\mathsf{HF\text{-}SETS\text{-}3} \vdash (\forall S, S', T : Set; M' : Magma)$$
$$\psi \Rightarrow augment(augment(\{S', M'\}, S), T) \downarrow augment(augment(\{S', M'\}, T), S)$$

where ψ is the formula:

$$augment(augment(\{S'\}, S), T) \downarrow augment(augment(\{S'\}, T), S) \wedge$$
$$augment(augment(\{M'\}, S), T) \downarrow augment(augment(\{M'\}, T), S).$$

For the first one of these two proof obligations, a proof can be found as follows:

```
search in HF-SETS-3-REACH :
  join(augment(augment({S'}, S), T), augment(augment({S'}, T), S)) =>! tt .
Solution 1 (state 14)
```

For the second proof obligation, it suffices to rewrite both terms in the consequent of the implication and use the second conjunct in ψ, together with the JOIN and CTX, to join the resulting terms:

```
search in HF-SETS-3-REACH : augment(augment({M',S'}, S), T) =>! X:Set .
Solution 1 (state 6)
X:Set --> {S' U {S,T}} U augment(augment({M'}, S), T)


search in HF-SETS-3-REACH : augment(augment({M',S'}, T), S) =>! X:Set .
Solution 1 (state 6)
X:Set --> {S' U {S,T}} U augment(augment({M'}, T), S)
```

Therefore, all critical pairs of HF-SETS-3 are ground joinable; hence, HF-SETS-3 is ground convergent, as desired.

But is the original specification HF-SETS itself ground convergent? That is, can the extra equations in HF-SETS-3 just be used as *scaffolding* and then be

removed as unnecessary? The following result shows that, if the successive addition of oriented equalities leads us to a ground convergent theory and such equalities are ground joinable, then the added equations are indeed unnecessary. The main idea is that, starting from an equational specification $\mathcal{E}_0$, if a sequence of equational theories $\mathcal{E}_0 \subseteq \mathcal{E}_1 \subseteq \cdots \subseteq \mathcal{E}_n$ can be built by incrementally adding new equations (e.g., suggested by the analysis of critical pairs between the equations), and if the new equations added at each step can be shown ground joinable, then the ground confluence of $\mathcal{E}_n$ implies the ground confluence of each $\mathcal{E}_i$, and in particular of $\mathcal{E}_0$.

Theorem 5. *Let $(\Sigma, E_0 \uplus B) \subseteq (\Sigma, E_1 \uplus B)$ where $\overrightarrow{E_0}, B$ is sufficiently complete with respect to a subsignature Ω, $(\Sigma, E_1 \uplus B)$ is ground convergent, $\rightarrow_{\overrightarrow{E_0},B} |_\Omega = \rightarrow_{\overrightarrow{E_1},B} |_\Omega$, and all equations in $E_1 - E_0$ are ground E_0, B-joinable. Then,*

$$\left(\rightarrow^!_{\overrightarrow{E_0},B}; =_B\right) |_{T_\Sigma} = \left(\rightarrow^!_{\overrightarrow{E_1},B}; =_B\right) |_{T_\Sigma}.$$

That is, the normal forms of the rewriting relation modulo B restricted to the initial term algebra T_Σ coincide.

Proof. First of all note that, since $\overrightarrow{E_0} \subseteq \overrightarrow{E_1}$, $(\Sigma, B, \overrightarrow{E_0})$ is operationally terminating. Consider some $t \in T_\Sigma$ and rewrite $t \rightarrow^!_{\overrightarrow{E_1},B} u$. Since $\overrightarrow{E_0}, B$ is sufficiently complete and $\rightarrow_{\overrightarrow{E_0},B} |_\Omega = \rightarrow_{\overrightarrow{E_1},B} |_\Omega$, $u \in T_\Omega$. If all rules applied in the chain are in $\overrightarrow{E_0}$, then the chains obviously coincide. Otherwise, let us consider the first rewrite step using a rule in $\overrightarrow{E_1} - \overrightarrow{E_0}$:

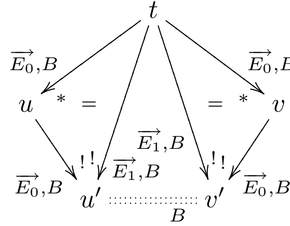
$$\begin{array}{ccccccc}
t \xrightarrow[\overrightarrow{E_0},B]{*} & \cdot & \xrightarrow[\overrightarrow{E_1}-\overrightarrow{E_0},B]{} & \cdot & \xrightarrow[\overrightarrow{E_1},B]{!} u \\
& {\scriptstyle \overrightarrow{E_0},B}\searrow^{!} & & \swarrow^{!}{\scriptstyle \overrightarrow{E_0},B} & \\
& & v =_B w & &
\end{array}$$

First, we have $v =_B w$ by ground joinability of equations in $E_1 - E_0$. Then, by the assumption that $\rightarrow_{\overrightarrow{E_0},B} |_\Omega =\rightarrow_{\overrightarrow{E_1},B} |_\Omega$, u and w are in E_1, B-canonical form, and by the ground confluence of $\overrightarrow{E_1}, B$ we must have $u =_B w$. Therefore, we can conclude that $\rightarrow^!_{\overrightarrow{E_0},B}; =_B |_{T_\Sigma} =\rightarrow^!_{\overrightarrow{E_1},B}; =_B |_{T_\Sigma}$, as desired. □

Theorem 6. *Suppose $(\Sigma, E_0 \uplus B) \subseteq \ldots \subseteq (\Sigma, E_n \uplus B)$, with $n \geq 0$, such that $\overrightarrow{E_0} \uplus B$ is sufficiently complete with respect to a subsignature Ω, $\overrightarrow{E_n} \uplus B$ is ground convergent, $(\rightarrow_{\overrightarrow{E_0},B})|_\Omega = (\rightarrow_{\overrightarrow{E_n},B})|_\Omega$, and all $E_{i+1} - E_i$ are ground E_i-joinable modulo B. Then, each $(\Sigma, E_i \uplus B)$ is ground convergent, for $0 \leq i \leq n$. Furthermore, all theories in the chain have the same initial algebra.*

Proof. By induction on n. It is trivial for $n = 0$. Suppose it true for n, and let us prove it true for $n+1$. Given a chain $(\Sigma, E_0 \uplus B) \subseteq (\Sigma, E_1 \uplus B) \subseteq \ldots \subseteq (\Sigma, E_n \uplus B)$, by the induction hypothesis—plus the fact that $(\Sigma, E_0 \uplus B)$ sufficiently

complete makes $(\Sigma, E_1 \uplus B)$ so as well—we get that $(\Sigma, E_1 \uplus B)$ is ground convergent. The proof that $(\Sigma, E_0 \uplus B)$ is ground convergent is as follows. Since $(\Sigma, E_1 \uplus B)$ is ground convergent, $(\Sigma, E_0 \uplus B)$ is *a fortiori* sort-decreasing and operationally terminating, so all we need to prove is its ground confluence. But since, by Theorem 5, $\rightarrow^!_{\overrightarrow{E_0},B}; =_B |_{T_\Sigma} = \rightarrow^!_{\overrightarrow{E_1},B}; =_B |_{T_\Sigma}$, the following diagram proves ground confluence of $\rightarrow_{\overrightarrow{E_0},B}$:



Note that $u' =_B v'$ by ground confluence of $\rightarrow_{\overrightarrow{E_1},B}$.

Finally, we already know by the Induction Hypothesis that all the theories

$$(\Sigma, E_1 \uplus B) \subseteq \cdots \subseteq (\Sigma, E_n \uplus B)$$

have the same initial algebra, and, by ground-joinability of $E_1 - E_0$, that

$$\mathcal{T}_{\Sigma/E_0 \uplus B} \models E_1 - E_0.$$

Therefore, we also get $\mathcal{T}_{\Sigma/E_0 \uplus B} = \mathcal{T}_{\Sigma/E_1 \uplus B}$, as desired. □

Theorem 6 justifies the view of the new equations suggested by critical pairs obtained, say, from the CRC tool, as hints for extending our original specification as "scaffolding" that can be abandoned *after* we have reached a ground convergent extension $(\Sigma, E_n \uplus B)$. Going back to the example in Sect. 3, once the HF-SETS-3 module has been proven ground convergent, we can conclude that the original HF-SETS specification is also ground convergent, *provided* we can show that the equations added at stage $i+1$ were ground joinable relative to stage i. This is shown to be the case in Sect. 5 by providing proofs of ground joinability for the five equations added in HF-SETS-0, HF-SETS-1, HF-SETS-2, and HF-SETS-3 in Sect. 3.

5 Ground Convergence of HF-SETS

The goal of this section is to conclude that the equational specification HF-SETS presented in Sect. 3 is ground convergent, and therefore that its initial model is a model of set theory without the axiom of infinity. The key tools for achieving this goal are the inference system for inductive joinability and Theorem 6, both presented in Sect. 4. By knowing that $\mathcal{R}_{\text{HF-SETS-3}}$ is terminating (see [10, Appendix B]), sort decreasing (Sect. 3), and that HF-SETS is sufficiently

complete (see [10, Appendix C]), the conditions in Theorem 6 apply and we just need to show the ground joinability of the added equations.

That is, since HF-SETS-3 is ground convergent and the theory inclusions

$$\textsf{HF-SETS} \subseteq \textsf{HF-SETS-0} \subseteq \textsf{HF-SETS-1} \subseteq \textsf{HF-SETS-2} \subseteq \textsf{HF-SETS-3}$$

satisfy the requirements of Theorem 6, it suffices to prove

$$\begin{aligned}
\textsf{HF-SETS} &\Vdash (\forall M : \mathit{Magma})\ M \in \{M\} \downarrow \mathit{true} \\
\textsf{HF-SETS} &\Vdash (\forall M, M' : \mathit{Magma})\ M \in \{M, M'\} \downarrow \mathit{true} \\
\textsf{HF-SETS-0} &\Vdash (\forall S : \mathit{Set})\ S \cup S \downarrow S \\
\textsf{HF-SETS-1} &\Vdash (\forall S, S', T : \mathit{Set})\ \mathit{augment}(S \cup S', T) \downarrow \mathit{augment}(S, T) \cup \mathit{augment}(S', T) \\
\textsf{HF-SETS-2} &\Vdash (\forall S, T : \mathit{Set})\ \mathit{augment}(\mathit{augment}(S, T), T) \downarrow \mathit{augment}(S, T)
\end{aligned}$$

in order to conclude that HF-SETS is ground convergent. In what follows, detailed proofs are provided for the last three proof obligations. The first two properties can be proved by following a similar approach.

The third proof obligation is dealt with by using the CtorCases rule on $S \in X_{\mathsf{Set}}$ with generating set $G_{\mathsf{Set}} = \{\{\}, \{M\}\}$ and $M \in X_{\mathsf{Magma}}$:

$$\begin{aligned}
\textsf{HF-SETS-0} &\Vdash \{\} \cup \{\} \downarrow \{\} \\
\textsf{HF-SETS-0} &\Vdash (\forall M : \mathit{Magma})\ \{M\} \cup \{M\} \downarrow \{M\}
\end{aligned}$$

These two proof obligations can be automatically discharged by Maude in $\mathcal{R}^{\approx}_{\textsf{HF-SETS-0}}$:

```
search in HF-SETS-0-REACH : join({} U {}, {}) =>! tt .
Solution 1 (state 2)


search in HF-SETS-0-REACH : join({M} U {M}, {M}) =>! tt .
Solution 1 (state 3)
```

Next, for the fourth proof obligation, several inference steps are needed. First, the CtorCases rule is used on S_{Set} with generating set $G_{\mathsf{Set}} = \{\{\}, \{M\}\}$ and $M \in X_{\mathsf{Magma}}$, resulting in the following proof obligations:

$$\begin{aligned}
\textsf{HF-SETS-1} &\Vdash (\forall S', T : \mathit{Set})\ \mathit{augment}(\{\} \cup S', T) \downarrow \mathit{augment}(\{\}, T) \cup \mathit{augment}(S', T) \\
\textsf{HF-SETS-1} &\Vdash (\forall S', T : \mathit{Set}; M : \mathit{Magma}) \\
&\qquad \mathit{augment}(\{M\} \cup S', T) \downarrow \mathit{augment}(\{M\}, T) \cup \mathit{augment}(S', T)
\end{aligned}$$

For the second one of these two proof obligations, the CtorCases rule on $S' \in X_{\mathsf{Set}}$ with generating set $H'_{\mathsf{Set}} = \{\{\}, \{M'\}\}$ and $M' \in X_{\mathsf{Magma}}$ is used; this transforms the second proof obligation in the following two proof obligations:

$$\begin{aligned}
\textsf{HF-SETS-1} &\Vdash (\forall T : \mathit{Set}; M : \mathit{Magma}) \\
&\qquad \mathit{augment}(\{M\} \cup \{\}, T) \downarrow \mathit{augment}(\{M\}, T) \cup \mathit{augment}(\{\}, T) \\
\textsf{HF-SETS-1} &\Vdash (\forall T : \mathit{Set}; M, M' : \mathit{Magma}) \\
&\qquad \mathit{augment}(\{M\} \cup \{M'\}, T) \downarrow \mathit{augment}(\{M\}, T) \cup \mathit{augment}(\{M'\}, T)
\end{aligned}$$

The remaining three proof obligations can be automatically discharged by Maude in $\mathcal{R}^{\approx}_{\mathsf{HF\text{-}SETS\text{-}1}}$ as follows:

```
search in HF-SETS-1-REACH : join(augment({} U S', T),
       augment({}, T) U augment(S', T)) =>! tt .
Solution 1 (state 6)
```

```
search in HF-SETS-1-REACH :
  join(augment({} U {M}, T), augment({}, T) U augment({M}, T)) =>! tt .
Solution 1 (state 6)
```

```
search in HF-SETS-1-REACH :
  join(augment({M} U {M'}, T), augment({M}, T) U augment({M'}, T)) =>! tt .
Solution 1 (state 3)
```

The fifth, and last proof obligation, is dealt with by using the CtorCases rule on $S \in X_{\mathsf{Set}}$ with generating set $G_{\mathsf{Set}} = \{\{\}, \{M\}\}$ and $M \in X_{\mathsf{Magma}}$. This rule application results in the following two proof obligations:

$$\begin{aligned} \mathsf{HF\text{-}SETS\text{-}2} &\Vdash (\forall T : Set)\ augment(augment(\{\}, T), T) \downarrow augment(\{\}, T) \\ \mathsf{HF\text{-}SETS\text{-}2} &\Vdash (\forall T : Set; M : Magma) \\ &\qquad augment(augment(\{M\}, T), T) \downarrow augment(\{M\}, T) \end{aligned}$$

The first proof obligation can be discharged automatically:

```
search in HF-SETS-2-REACH : join(augment(augment({}, T), T), augment({}, T)) =>! tt .
Solution 1 (state 4)
```

The remaining proof obligation can be handled with the help of the GSInd rule with generating set $G_{\mathsf{Magma}} = \{S', (S', M')\}$, $S' \in X_{\mathsf{Set}}$ and $M' \in X_{\mathsf{Magma}}$:

$$\begin{aligned} \mathsf{HF\text{-}SETS\text{-}2} &\Vdash (\forall S', T : Set)\ augment(augment(\{S'\}, T), T) \downarrow augment(\{S'\}, T) \\ \mathsf{HF\text{-}SETS\text{-}2} &\Vdash (\forall\ S', T : Set; M' : Magma) \\ &\qquad \psi \Rightarrow augment(augment(\{S', M'\}, T), T) \downarrow augment(\{S', M'\}, T) \end{aligned}$$

where ψ is the formula:

$$\begin{aligned} & augment(augment(\{S'\}, T), T) \downarrow augment(\{S'\}, T) \\ \wedge\ & augment(augment(\{M'\}, T), T) \downarrow augment(\{M'\}, T). \end{aligned}$$

These two proof obligations can be solved with the help of Maude:

```
search in HF-SETS-2-REACH : join(augment(augment({S'}, T), T), augment({S'}, T)) =>! tt .
Solution 1 (state 10)
```

```
search in HF-SETS-2-REACH : augment(augment({M',S'}, T), T) =>! X:Set .
Solution 1 (state 7)
X:Set --> {S' U {T}} U augment(augment({M'}, T), T)
```

```
search in HF-SETS-2-REACH : augment({M',S'}, T) =>! X:Set .
Solution 1 (state 2)
X:Set --> {S' U {T}} U augment({M'}, T)
```

Note that the terms obtained by the last two search commands can be joined by assuming ψ.

The initial goal has now been reached. Namely, since all the equations added in the process of building the tower of theory inclusions

$$\mathsf{HF\text{-}SETS} \subseteq \mathsf{HF\text{-}SETS\text{-}0} \subseteq \mathsf{HF\text{-}SETS\text{-}1} \subseteq \mathsf{HF\text{-}SETS\text{-}2} \subseteq \mathsf{HF\text{-}SETS\text{-}3}$$

have been shown ground joinable, Theorem 6 guarantees that the equational specification HF-SETS for hereditarily finite sets is ground convergent.

6 Related Work and Conclusion

In [2], Bouhoula proposes an inference system for simultaneously checking the sufficient completeness and ground confluence of constructor-based equational specifications. His approach computes a pattern tree for every defined symbol and identifies a set of proof obligations whose inductive validity has to be checked: if they all are inductive theorems, then the specification is both sufficiently complete and ground confluent; otherwise, it outputs a counterexample. The main difference between the two approaches is that the one presented in this paper can handle both conditional specifications and reasoning modulo axioms, while [2] does not support reasoning modulo axioms. More recently, Nakamura et al. [17] propose incremental techniques for proving termination, confluence, and sufficient completeness of OBJ specifications. Their inference system is also based on the notion of constructor subsignatures, handles conditional equations, and provides sufficient conditions for ensuring such an incremental extension in a modular way. However, for ground confluence, their method has been developed for extensions that preserve the set of critical pairs relative to the extended specification.

Different tools and techniques have been proposed for proving and disproving confluence. Tools such as CSI [16] or ACP [1] are automatic confluence provers for first-order rewrite systems. These tools implement different criteria for proving both confluence and non-confluence.

This work has addressed a thorny and important problem in reasoning about equational programs and algebraic specifications with an initial algebra semantics: the fact that in practice a substantial number of such programs and specifications are *perfectly reasonable* and there is nothing wrong with them, yet they are not locally confluent and therefore fall outside the scope of the standard methods to prove them ground convergent. As the HF-SETS example has shown, it is quite mistaken to assume that, since our program is perfectly reasonable, we should be able to complete it in some Knuth-Bendix-like fashion, because we can easily hit a non-orientability "wall." We have proposed a general methodology to help verify the ground convergence of a given equational program in such a way that: (i) the *heuristic* value of using unjoinable critical pairs as *hints* is preserved; (ii) we can break through the wall of non-orientable equations by means of *inductive joinability* proof methods; and (iii) we can prove that our *original* specification is ground convergent and that its initial algebra semantics has been

preserved by its subsequent extensions using the same inductive joinability proof techniques.

Future work suggested by this work includes: (i) full mechanization of the inductive joinability inference system and its integration within the Maude Formal Environment; (ii) further experimentation with these methods on a rich collection of examples; and (iii) development of new proof techniques complementing those presented here.

Acknowledgments. The authors would like to thank the anonymous referees for their helpful comments that helped us improve the paper. The first author was partially supported by Spanish MINECO/FEDER project TIN2014-52034-R and Univ. Málaga, Campus de Excelencia Internacional Andalucía Tech. The second author was partially supported by NRL under contract number N00173-17-1-G002. The third author was partially supported by CAPES, Colciencias, and INRIA via the STIC AmSud project "EPIC: EPistemic Interactive Concurrency" (Proc. No 88881.117603/2016-01).

References

1. Aoto, T., Yoshida, J., Toyama, Y.: Proving confluence of term rewriting systems automatically. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 93–102. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02348-4_7
2. Bouhoula, A.: Simultaneous checking of completeness and ground confluence for algebraic specifications. ACM Trans. Comput. Log. **10**(3), 1–33 (2009)
3. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theor. Comput. Sci. **360**(1–3), 386–414 (2006)
4. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
5. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 243–320. North-Holland, Amsterdam (1990)
6. Durán, F., Lucas, S., Marché, C., Meseguer, J., Urbain, X.: Proving operational termination of membership equational programs. High.-Order Symb. Comput. **21**(1–2), 59–88 (2008)
7. Durán, F., Lucas, S., Meseguer, J.: MTT: the Maude termination tool (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 313–319. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_27
8. Durán, F., Lucas, S., Meseguer, J.: Termination modulo combinations of equational theories. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS (LNAI), vol. 5749, pp. 246–262. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04222-5_15
9. Durán, F., Meseguer, J.: On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. J. Log. Algebr. Program. **81**(7–8), 816–850 (2012)
10. Durán, F., Meseguer, J., Rocha, C.: Proving ground confluence of equational specifications modulo axioms. Technical report 2142/99548, University of Illinois, Urbana, USA, March 2018

11. Durán, F., Rocha, C., Álvarez, J.M.: Towards a Maude formal environment. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 329–351. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_17
12. Goguen, J.A., Meseguer, J.: Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci. **105**(2), 217–273 (1992)
13. Hrbacek, K., Jech, T.J.: Introduction to Set Theory. Monographs and Textbooks in Pure and Applied Mathematics, vol. 45, 3rd edn. M. Dekker, New York (1999)
14. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Presicce, F.P. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-64299-4_26
15. Meseguer, J.: Strict coherence of conditional rewriting modulo axioms. Theor. Comput. Sci. **672**, 1–35 (2017)
16. Nagele, J., Felgenhauer, B., Middeldorp, A.: CSI: new evidence – a progress report. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 385–397. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_24
17. Nakamura, M., Ogata, K., Futatsugi, K.: Incremental proofs of termination, confluence and sufficient completeness of OBJ specifications. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 92–109. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54624-2_5
18. Rocha, C., Meseguer, J.: Constructors, sufficient completeness, and deadlock freedom of rewrite theories. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR 2010. LNCS, vol. 6397, pp. 594–609. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16242-8_42

Uniform Strong Normalization for Multi-discipline Calculi

Paul Downen(✉), Philip Johnson-Freyd, and Zena M. Ariola

University of Oregon, Eugene, USA
{pdownen,philipjf,ariola}@cs.uoregon.edu

Abstract. Modern programming languages have effects and mix multiple calling conventions, and their core calculi should too. We characterize calling conventions by their "substitution discipline" that says what variables stand for, and design calculi for mixing disciplines in a single program. Building on variations of the reducibility candidates method, including biorthogonality and symmetric candidates which are both specialized for one discipline, we develop a single uniform framework for strong normalization encompassing call-by-name, call-by-value, call-by-need, call-by-push-value, non-deterministic disciplines, and any others satisfying some simple criteria. We explicate commonalities of previous methods and show they are special cases of the uniform framework and they extend to multi-discipline programs.

1 Introduction

Picking a programming language means choosing not just a concrete syntax and set of features, but also a calling convention. As Simon Peyton Jones [19] says:

> These days, the strict/lazy decision isn't a straight either/or choice. For example, a lazy language has ways of stating "use call-by-value here," and even if you were to say "Oh, the language should be call by value," you would want ways to achieve laziness anyway. Any successor language to Haskell will have support for both strict and lazy functions. So the question then is: "How do you mix them together?"

This question is as important in language theory as it is in practice: different programming languages merit different calculi. For example, just $\beta\eta$ axioms are enough for equality of call-by-name functions, but more axioms are needed to complete the theory of call-by-value [10,24]. More drastically, call-by-need requires some extra rules even for computing answers. If we then want to reflect the reality of programming languages that mix calling conventions, we need a theory that mixes them, too. Again, the question is: "How?"

Polarized logic [18,30] and call-by-push-value [15] partially answers the question of how to mix calling conventions by dividing types into two groups: positive and negative. The positive types, like sums, follow the call-by-value discipline whereas the negative types, like functions, follow the call-by-name regime.

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 205–225, 2018.
https://doi.org/10.1007/978-3-319-99840-4_12

Here, by contrast, we do connect calling conventions with types, but allow each type constructor to build a type of any convention; for example we can have both a call-by-value or a call-by-need function type. This more closely reflects practice where OCaml has call-by-value functions.

Even though each calculus for each convention is different, they can be all be seen as variations on the same idea. As pioneered by Ronchi Della Rocca *et al.* [23], calculi for different calling conventions can be summarized as instances of a common calculus parameterized by a *substitution discipline* [4] specifying what might be substituted for identifiers. Call-by-name and -value can then share the same $\beta\eta$ axioms, $(\lambda x.M)V = M\{V/x\}$ and $\lambda x.V x = V$; what changes is the notion of *value* V. Call-by-name says that V can be any term, and call-by-value is more restrictive. Each of the above mentioned three calling conventions can be uniformly represented, as well as more exotic ones like the dual to call-by-need [1] and the non-deterministic evaluation of the symmetric λ-calculus [2].

Abstracting away the differences across languages enables us to study properties of those languages in a uniform way. In this paper, we focus on strong normalization. Currently, there are separate proofs of strong normalization for calculi of different disciplines. Here, we show *one* common proof for all of them by articulating the essential properties of the substitution discipline that guarantees strong normalization. We build on a technique previously used for studying a language of mixed induction and co-induction [5], which is based on both biorthogonal [7,13,21] and symmetric candidate [2] models, and extend it to accommodate multi-discipline languages. Furthermore, the more refined version of the technique presented here lets us formally understand the relationship between orthogonality and symmetric candidates: biorthogonality models are subsumed as a special case of our uniform model.

The orthogonality-based family of methods require that we not only think of how to create values of a type, but also how to use them. This inevitably leads to the invention of abstract-machine-like constructs to represent a reified environment or context of a program fragment [13,21]. Instead of going about an ad hoc reification, we base our proof on a classical sequent calculus which is already an abstract machine language (Sect. 2) that is well-suited to mixing disciplines (Sect. 3).

This work uses a sequent calculus with impredicative polymorphism based on [5] and extended with multiple disciplines—which are given as a parameter to the system and not fixed a priori—in the sense that different calling conventions can be used in the same program (Sect. 4). Our contributions are:

- A uniform proof of strong normalization based on orthogonality and symmetric candidates that parametrically accounts for multiple disciplines (Sect. 5).
- A more precise model than [5] which subsumes biorthogonality models for call-by-name, -value, and -push-value as special instances, and the first proof of strong normalization for multi-discipline call-by-need and its dual (Sect. 6).

The proofs for Sects. 4 to 6 together with a strongly normalizing and polymorphic $\lambda\mu$-calculus that mixes call-by-value, -name, and -need are given in the

extended version of this paper which can be accessed at http://ix.cs.uoregon.edu/~pdownen/publications/wrla18.pdf.

2 A Language Approach to Abstract Machines

One of the most basic ways of evaluating a λ-calculus term is by repeated β reduction. For instance, if we have the term $(\lambda x.\lambda y.x + y)\ 1\ 2$ we can compute a value in three steps:

$$(\lambda x.\lambda y.x + y)\ 1\ 2 \to (\lambda y.1 + y)\ 2 \to 1 + 2 \to 3$$

However, even in this simple example we can observe one frustration with the β-reduction model from the perspective of implementation: reductions might not always occur at the "top" of the term, but can be buried somewhere within it. In the very first reduction step above, the redex $(\lambda x.\lambda y.x + y)\ 1$ subjected to β reduction happens inside of the outermost application context $\Box\ 2$, where $\Box$ stands for the position of the sub-term within the context. As such, performing evaluation by β reduction requires a search for the next redex within a term, which must be specified as part of an implementation of the evaluator.

An *abstract machine* gives a lower-level description of evaluation by interweaving search and reduction together. To keep track of its position within the term, a machine does not evaluate terms directly but rather larger configurations. Here, the configurations we use are called *commands* (denoted by the metavariable c) which consist of a term (denoted by v) together with a syntactic representation of its context called *co-term* (denoted by e). One abstract machine in this style is the Krivine machine [12], which requires only two rules:

$$\langle v\ v' \| e\rangle \to \langle v \| v' \cdot e\rangle \qquad\qquad \langle \lambda x.v \| v' \cdot e\rangle \to \langle v\{v'/x\} \| e\rangle$$

The first rule pushes the argument of a function call onto the call-stack. In other words, evaluating an application of the form $v\ v'$ in a surrounding context e consists of pushing the argument v' on top of e and then evaluating v in the larger context. The second rule implements β reduction by popping the top argument off of a call-stack and plugging it into the formal parameter of a λ-abstraction. In the Krivine machine style, our previous example can be computed as follows, where we assume the term is evaluated in a context named α:

$$\begin{aligned}\langle(\lambda x.\lambda y.x + y)\ 1\ 2 \| \alpha\rangle &\to \langle(\lambda x.\lambda y.x + y)\ 1 \| 2 \cdot \alpha\rangle \\ &\to \langle\lambda x.\lambda y.x + y \| 1 \cdot 2 \cdot \alpha\rangle \\ &\to \langle\lambda y.1 + y \| 2 \cdot \alpha\rangle \to \langle 1 + 2 \| \alpha\rangle \to \langle 3 \| \alpha\rangle\end{aligned}$$

So the machine returns same result, 3, to the surrounding context as was achieved by β reduction. The Krivine machine thus seems to represent a lower level implementation, one closer to actual computation on a physical machine using call-stacks. Moreover, exploring the laws of the Krivine machine suggests additional possibilities. We see in the Krivine machine that there are actually two different

syntactic constructs for invoking a function: both configurations $\langle \square \parallel v \cdot e\rangle$ and $\langle \square\ v \parallel e\rangle$ do exactly the same thing as the second is rewritten into the first. That is, both call-stack formation and ordinary λ calculus application are two ways of getting at the same concept. It is thus natural to wonder if the redundancy can be eliminated by unifying the two.

We are accustomed to having variables stand for an unknown value and then having the possibility to bind these variables to known terms later. The same can be done with respect to contexts, now that they are embodied with a syntactic representation in the form of co-terms. Already in the example above we refer to α (called a co-variable) as a generic placeholder for the surrounding context of evaluation. The next is to abstract over co-variables like α. That is the role of the μ-abstraction, written as $\mu\alpha.c$, which is reduced like so:

$$\langle \mu\alpha.c \parallel e\rangle \to c\{e/\alpha\}$$

The above says that when the term $\mu\alpha.c$ is evaluated in a context e, then the next step is to execute the command c with α bound to e. μ-abstractions unify the two forms of function calls by representing function application in terms of call-stack formation. For example, the above λ-calculus term $(\lambda x.\lambda y.x+y)1\ 2$ can be rewritten to avoid function application altogether as $\mu\beta.\langle \lambda x.\lambda y.x+y \parallel 1\cdot 2\cdot \beta\rangle$. Note that this term behaves the same as the original one:

$$\langle \mu\beta.\langle \lambda x.\lambda y.x+y \parallel 1\cdot 2\cdot \beta\rangle \parallel \alpha\rangle \to \langle \lambda x.\lambda y.x+y \parallel 1\cdot 2\cdot \alpha\rangle$$

As such, the application term $v\ v'$ becomes syntactic sugar for $\mu\alpha.\langle v \parallel v'\cdot \alpha\rangle$.

However, the presence of μ-abstraction makes the language more expressive than λ-calculus because a μ has the ability to *erase* its context when the abstracted co-variable is never used:

$$\langle \mu\beta.\langle \lambda x.\lambda y.x+y \parallel \alpha\rangle \parallel 1\cdot 2\cdot \alpha\rangle \to \langle \lambda x.\lambda y.x+y \parallel \alpha\rangle$$

A μ-abstraction can also *duplicate* its context by using the abstracted co-variable more than once. Indeed, terms such as $\mu\alpha.c$ create a *control* effect much like those found in many programming languages. In particular, μ-abstractions are similar to the `callcc` operator from Scheme.

So far, this analysis gives rise to a language for representing abstract machines implementing call-by-name evaluation. But what about call-by-value evaluation, where arguments are evaluated before resolving a function application, giving rise to evaluation contexts of the form $V\ \square$ (where V denotes a value: a variable or a λ-abstraction) in addition to $\square\ v$. The call-by-value version of the above Krivine machine would use an extra co-term $V \circ e$ corresponding to the additional form of evaluation context (first apply V to the input and return the result to e), as well as the following three reduction rules:

$$\langle v\ v' \parallel e\rangle \to \langle v \parallel v'\cdot e\rangle \quad \langle V \parallel v'\cdot e\rangle \to \langle v' \parallel V\circ e\rangle \quad \langle V' \parallel (\lambda x.v)\circ e\rangle \to \langle v\{V'/x\} \parallel e\rangle$$

The first rule pushes an argument onto the call-stack as before. The second rule switches the attention of the machine from the function, represented by V,

to the argument v' beginning evaluation of the argument by placing it on the left-hand side of the command. The third rule implements β reduction slightly differently from before, since the function is now found in the co-term after evaluation due to the second rule. The call-by-value evaluation of our example above becomes:

$$\begin{aligned}\langle(\lambda x.\lambda y.x + y)\ 1\ 2 \parallel \alpha\rangle &\twoheadrightarrow \langle\lambda x.\lambda y.x + y \parallel 1 \cdot 2 \cdot \alpha\rangle \\ &\rightarrow \langle 1 \parallel (\lambda x.\lambda y.x + y) \circ (2 \cdot \alpha)\rangle \\ &\rightarrow \langle\lambda y.1 + y \parallel 2 \cdot \alpha\rangle \\ &\rightarrow \langle 2 \parallel (\lambda y.1 + y) \circ \alpha\rangle \rightarrow \langle 1 + 2 \parallel \alpha\rangle \rightarrow \langle 3 \parallel \alpha\rangle\end{aligned}$$

Besides changing the language of co-terms to account for a different evaluation strategy, this presentation of call-by-value machines suffers even worse redundancy: there are *three* different syntactic representations of function invocation—$\langle(\lambda x.v)\ v' \parallel e\rangle$, $\langle\lambda x.v \parallel v' \cdot e\rangle$, and $\langle v' \parallel \lambda x.v \circ e\rangle$—all of which are equivalent to one another. In the interest of eliminating redundancy, we should again wonder if all notions of function invocation can be distilled down to a single primitive operation with the help of some other generic binding constructs, like μ. Indeed, call-by-value can employ the *dual* of μ-abstractions, known as $\tilde{\mu}$-abstractions [3], to write everything with call-stacks. Symmetric to a μ, the $\tilde{\mu}$-abstraction $\tilde{\mu}x.c$ is a co-term that binds its input to the variable x and then runs the command c:

$$\langle v \parallel \tilde{\mu}x.c\rangle \rightarrow c\{v/x\}$$

Just like μ-abstractions can be used to write a λ-calculus application with a call-stack, so too can $\tilde{\mu}$-abstractions be used to write the extra call-by-value evaluation context with the primitive form of call-stack: $v \circ e$ becomes syntactic sugar for $\tilde{\mu}x.\langle v \parallel x \cdot e\rangle$. Expanding this notational definition, the second rule is:

$$\langle V \parallel v' \cdot e\rangle \rightarrow \langle v' \parallel \tilde{\mu}x.\langle V \parallel x \circ e\rangle\rangle$$

which names the argument for evaluation, and the call-by-value implementation of β reduction simplifies to the call-by-name one:

$$\langle V' \parallel (\lambda x.v) \circ e\rangle = \langle V' \parallel \tilde{\mu}y.\langle\lambda x.v \parallel y \cdot e\rangle\rangle \rightarrow \langle\lambda x.v \parallel V' \cdot e\rangle \rightarrow \langle v\{V'/x\} \parallel e\rangle$$

A Calculus for Abstract Machines. These basic constructs—functions and call-stacks, variables and co-variables, μ- and $\tilde{\mu}$-abstractions—define a general calculus for reasoning about abstract machines (both call-by-value and call-by-name) known as system L [17]. System L is a lower-level machine-like calculus, in that no search is needed for evaluation: reduction can always take place at the "top" of a command. But system L also supports high-level reasoning like the λ-calculus, in that it is still *sound* to perform reductions anywhere within a command, which correspond to out-of-order simplifications and optimizations. Also like the λ-calculus, system L can be seen as either an

untyped or typed language. Since there are two different forms of variables—both ordinary variables and co-variables—there are two typing environments: $\Gamma = x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$ for tracking the types of free variables and $\Delta = \alpha_1 : A_1, \alpha_2 : A_2, \ldots, \alpha_n : A_n$ for tracking the types of free co-variables. Since there are three different forms of expressions—commands, terms, and co-terms—there are three different typing judgements. Terms returning a result of type A in environments Γ and Δ are typed as $\Gamma \vdash v : A \mid \Delta$. Co-terms expecting an input of type A in environments Γ and Δ are typed as $\Gamma \mid e : A \vdash \Delta$. And commands that are capable of running in environments Γ and Δ are typed as $c : (\Gamma \vdash \Delta)$. With this notation in mind, the typing rules for the L-style language of abstract machines are:

$$\frac{\Gamma, x{:}A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda x.v : A \to B \mid \Delta} \qquad \frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid v \cdot e : A \to B \vdash \Delta}$$

$$\frac{}{\Gamma, x{:}A \vdash x : A \mid \Delta} \quad \frac{c : (\Gamma \vdash \alpha{:}A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta} \quad \frac{c : (\Gamma, x{:}A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta} \quad \frac{}{\Gamma \mid \alpha{:}A \vdash \alpha : A, \Delta}$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle v \| e \rangle : (\Gamma \vdash \Delta)}$$

Amazingly, in the same way that the typing rules for λ-calculus correspond to the rules of natural deduction, the above typing rules correspond to the sequent calculus [3]! The typing rules for call-stacks and commands correspond to the logical rules for implication (on the left) and cut. λ-abstractions are typed as usual, the two axioms correspond to (co-)variables, and the $\mu\tilde{\mu}$ abstractions allow one to focus on an assumption or conclusion.

3 Substitution Disciplines

But there is a problem that rears its head when we try to compute; the fundamental critical pair of classical logic between the μ- and $\tilde{\mu}$-abstractions [3]:

$$c_1\{\tilde{\mu}x.c_2/\alpha\} \leftarrow \langle \mu\alpha.c_1 \| \tilde{\mu}x.c_2 \rangle \to c_2\{\mu\alpha.c_1/x\}.$$

The choice between these two reductions takes us down two separate paths. In the worst case, x and α are never used and c_1 and c_2 are unrelated to one another, which means that a single command can reduce to two completely unrelated results. This critical pair can be resolved by always preferring one reduction or the other, giving two different calculi. Favoring μ by always taking the left path gives the call-by-value calculus, whereas favoring $\tilde{\mu}$ by always taking the right path gives the call-by-name calculus.

As observed by Plotkin [22], different calling conventions require different calculi: the traditional λ-calculus is suitable for reasoning about Haskell programs, as the call-by-value λ-calculus is for OCaml programs. But denotational semantics seems to capture the essential difference between call-by-name and call-by-value more generally: the difference is reflected in the *Denotable* domain

[26]. A call-by-name variable can denote any expressible value, including errors or divergence, whereas a call-by-value variable can only denotes "regular" values.

This idea can be represented syntactically by characterizing the calculus in two parts [4,23]; one part is common to different parameter passing techniques and the other only differs in one aspect: what can be substituted for a variable and co-variable. We refer to what variables and co-variables stand for as a *substitution discipline*. We call a term that can be substituted for a variable a *value*, and call a co-term that can be substituted for a co-variable a *co-value*. Thus, the call-by-name calculus is defined by saying that *every* term is a substitutable value, while the set of co-values is restricted to the bare minimum necessary to not get stuck. Symmetrically, the call-by-value calculus is formed by saying that *every* co-term is a co-value, and restricting values down to the bare minimum to avoid getting stuck. Moreover, call-by-name and -value are not the only disciplines expressible in this framework. For instance, call-by-need can be characterized by the notion of substitution discipline as well [1].

Mixing Disciplines. This framework allows for a characterization of the differences between calling conventions as a resolution to the above fundamental critical pair, which can be further distilled into a discipline on substitution. Why, then, should only choose one discipline globally for the entire program? Often times such a restriction can be quite limiting. As observed in [20], some functions like $\lambda x.x + x$ will always evaluate their argument eagerly even in a lazy language, and as such the extra costs associated with lazy evaluation should be avoided when laziness is irrelevant. Thus, it would be more practical to let the programmer, or at least the compiler during code generation and optimization, choose which discipline is appropriate for each juncture. In other words, we want a *multi-discipline* language that incorporates many calling conventions.

The obvious way to signal the intended discipline is to just annotate each command with symbols such as $\mathbf{v}$ (for call-by-value) and $\mathbf{n}$ (for call-by-name), which resolves the fundamental critical pair on a per-command basis. So in the above example, we could write the call-by-value choice as $\langle \mu\alpha.c_1|\mathbf{v}|\tilde{\mu}x.c_2\rangle \to c_1\{\tilde{\mu}x.c_2/\alpha\}$ and the call-by-name choice as $\langle \mu\alpha.c_1|\mathbf{n}|\tilde{\mu}x.c_2\rangle \to c_2\{\mu\alpha.c_1/x\}$. Unfortunately, just marking commands is not en ough, as it only pushes the issue of the critical pair one step away. The problem is that we could *lie* about what a variable or co-variable denotes by using it in a context that violates the contract of its binding. For example, the same critical pair is simulated as follows:

$$\langle \mu\alpha.c_1|\mathbf{v}|\tilde{\mu}y.c_2\rangle \leftarrow \langle \mu\alpha.c_1|\mathbf{n}|\tilde{\mu}x.\,\langle x|\mathbf{v}|\tilde{\mu}y.c_2\rangle\rangle \to \langle \mu\alpha.c_1|\mathbf{n}|\tilde{\mu}x.c_2\{x/y\}\rangle\,.$$

By reducing the top redex and plugging in the computation $\mu\alpha.c_1$ for the $\mathbf{n}$ variable x, on the left we end up with a $\mathbf{v}$ command that will prioritize the term. But by instead performing the inner redex, we end up with the equivalent $\mathbf{n}$ command that will prioritize the co-term.

So a multi-discipline sequent calculus cannot just annotate commands, but must ensure that the chosen discipline of variables and co-variables remains consistent throughout their lifetime. To make this choice apparent in the syntax,

variables and co-variables must have a statically-inferable discipline which we accomplish with annotations, *e.g.,* $x^{\mathbf{v}}$ and $\alpha^{\mathbf{n}}$. Furthermore, terms and co-terms in general also much have a statically-inferable discipline, since it is sometimes necessary to introduce a new binding during reduction. For example, recall the second rule of the call-by-value abstract machine in Sect. 2, which corresponds to naming the argument of a function with a $\tilde{\mu}$-abstraction. This naming step is necessary to avoid getting stuck during a call-by-value function call: call-by-value β reduction does not apply to $\langle \lambda x.v \parallel v' \cdot e\rangle$ when v' is not a value. This is done by *lifting* v' out of the call-stack [4]: $\langle \lambda x^{\mathbf{v}}.v \parallel v' \cdot e\rangle \to \langle \lambda x^{\mathbf{v}}.v \parallel \tilde{\mu}x.\langle v' \parallel \tilde{\mu}y.\langle x \parallel y \cdot e\rangle\rangle\rangle$. However, to annotate α and y above, we would need to know what the intended disciplines of $\lambda x^{\mathbf{v}}.v$ and e are.

4 A Parametric, Multi-discipline Sequent Calculus

We now formalize the core calculus for studying multi-discipline reduction in the presence of control. For simplicity we limit to a few key type formers: functions and parametric polymorphism. These features are found in most real functional programming languages, are enough both to write a variety of interesting programs, and expose the main challenges faced in strong normalization proofs.

$$\mathbf{r}, \mathbf{s}, \mathbf{t} \in \mathit{Kind} ::= \mathbf{n} \mid \mathbf{v} \mid \ldots$$

$$A_{\mathbf{s}}, B_{\mathbf{s}} \in \mathit{Type}_{\mathbf{s}} ::= a^{\mathbf{s}} \mid A \xrightarrow{\mathbf{s}} B \mid \forall^{\mathbf{s}} \boldsymbol{a}.A \qquad A, B \in \mathit{Type} ::= A_{\mathbf{s}}$$

$$v_{\mathbf{s}} \in \mathit{Term}_{\mathbf{s}} ::= x^{\mathbf{s}} \mid \mu\alpha^{\mathbf{s}}.c \mid \mu(\boldsymbol{x} \cdot_{\mathbf{s}} \boldsymbol{\alpha}).c \mid \mu(\boldsymbol{a} \cdot_{\mathbf{s}} \boldsymbol{\alpha}).c$$

$$e_{\mathbf{s}} \in \mathit{Co\text{-}Term}_{\mathbf{s}} ::= \alpha^{\mathbf{s}} \mid \tilde{\mu}x^{\mathbf{s}}.c \mid v \cdot_{\mathbf{s}} e \mid A \cdot_{\mathbf{s}} e$$

$$c \in \mathit{Command} ::= \langle v_{\mathbf{s}} \| e_{\mathbf{s}}\rangle \qquad v \in \mathit{Term} ::= v_{\mathbf{s}} \qquad e \in \mathit{Co\text{-}Term} ::= e_{\mathbf{s}}$$

$$\boldsymbol{x} ::= x^{\mathbf{t}} \qquad \boldsymbol{a} ::= a^{\mathbf{t}} \qquad \boldsymbol{\alpha} ::= \alpha^{\mathbf{t}}$$

Fig. 1. Syntax of a multi-discipline, polymorphic sequent calculus.

Syntax. As in the abstract machine language of Sect. 2, the syntax of our calculus is comprised of terms ("producers" v), co-terms ("consumers" e), and commands ("executables" c) as shown in Fig. 1. The first thing to notice is a change of syntax for functions. Instead of λ-abstractions, functions are written by *pattern-matching* on their context: a call-stack of the form $x \cdot \alpha$. This change of notation is syntactic in nature—note that $\lambda x.v$ is equivalent to $\mu(x \cdot \alpha).\langle v \parallel \alpha\rangle$—which helps to emphasize the role of functions as responders to call-stacks. As in system F, polymorphism is expressed in terms of type abstraction and specialization. Note that these constructs are analogous to functions, except that the parameter is a type, not a value.

The second thing to notice about the syntax is that terms and co-terms are divided by their discipline as discussed in Sect. 3, a finite collection of symbols denoted by the metavariable $\mathbf{s}$, so that $v_{\mathbf{s}}$ produces an $\mathbf{s}$ value and $e_{\mathbf{s}}$ consumes an $\mathbf{s}$ value. This aligns with the annotations on variables and co-variables,

where $x^{\mathbf{s}}$ is a member of (only) $Term_{\mathbf{s}}$ and similarly $\alpha^{\mathbf{s}}$ is in $Co\text{-}Term_{\mathbf{s}}$. A bold (co-)variable denotes an annotated (co-)variable, respectively, where the annotation could be any discipline. Commands, in contrast, do not have an outwardly-visible discipline because they do not produce or consume anything, but instead are only well-formed if they have an internally-consistent discipline shared by a producer and consumer cooperating together. To ensure that *every* term and co-term belong to exactly one syntactic category $Term_{\mathbf{s}}$ and $Co\text{-}Term_{\mathbf{s}}$, the call-stack dot is also annotated with a discipline symbol. That way, it is immediately apparent that $v \textcircled{s} e$ is an $\mathbf{s}$ co-term and $\mu(\boldsymbol{x} \textcircled{s} \boldsymbol{\alpha}).c$ is an $\mathbf{s}$ term. For example, a wholly call-by-value function can be written as $\mu(x^{\mathbf{v}} \textcircled{v} \alpha^{\mathbf{v}}).c$ that matches a call-stack of the form $v_{\mathbf{v}} \textcircled{v} e_{\mathbf{v}}$. The $\mathbf{v}$ in the $\textcircled{v}$ tells us the discipline used for computing the function itself, whereas the annotations on the abstracted (co-)variables tell us the discipline of the argument and result. Replacing $\mathbf{v}$ with $\mathbf{n}$ gives instead wholly call-by-name functions, but other more interesting combinations are also possible. The functions found in call-by-push-value [15] and polarized languages [30] would have the form $\mu(x^{\mathbf{v}} \textcircled{n} \alpha^{\mathbf{n}}).c$ and $v_{\mathbf{v}} \textcircled{n} e_{\mathbf{n}}$, with a call-by-value argument and call-by-name function and result.

$$\langle \mu\boldsymbol{\alpha}.c \| E\rangle \succ_{\mu} c\{E/\boldsymbol{\alpha}\} \qquad \langle V \| \tilde{\mu}\boldsymbol{x}.c\rangle \succ_{\tilde{\mu}} c\{V/\boldsymbol{x}\} \qquad \mu\boldsymbol{\alpha}.\langle v\|\boldsymbol{\alpha}\rangle \succ_{\eta_\mu} v \qquad \tilde{\mu}\boldsymbol{x}.\langle \boldsymbol{x}\|e\rangle \succ_{\eta_{\tilde{\mu}}} e$$

$$\langle \mu(x^{\mathbf{t}} \textcircled{s} \alpha^{\mathbf{r}}).c \| V_{\mathbf{t}} \textcircled{s} E_{\mathbf{r}}\rangle \succ_{\beta^{\to}} c\{V_{\mathbf{t}}/x^{\mathbf{t}}, E_{\mathbf{r}}/\alpha^{\mathbf{r}}\}$$

$$\langle \mu(a^{\mathbf{t}} \textcircled{s} \alpha^{\mathbf{r}}).c \| A_{\mathbf{t}} \textcircled{s} E_{\mathbf{r}}\rangle \succ_{\beta^{\forall}} c\{A_{\mathbf{t}}/a^{\mathbf{t}}, E_{\mathbf{r}}/\alpha^{\mathbf{r}}\}$$

$$v_{\mathbf{t}} \textcircled{s} e \succ_{\varsigma^{\to}} \tilde{\mu}x^{\mathbf{s}}.\langle v_{\mathbf{t}} \| \tilde{\mu}y^{\mathbf{t}}.\langle x^{\mathbf{s}} \| y^{\mathbf{t}} \textcircled{s} e\rangle\rangle \qquad (\nexists V_{\mathbf{t}}, v_{\mathbf{t}} = V_{\mathbf{t}})$$

$$V \textcircled{s} e_{\mathbf{r}} \succ_{\varsigma^{\to}} \tilde{\mu}x^{\mathbf{s}}.\langle \mu\beta^{\mathbf{r}}.\langle x^{\mathbf{s}} \| V \textcircled{s} \beta^{\mathbf{s}_2}\rangle \| e_{\mathbf{r}}\rangle \qquad (\nexists E_{\mathbf{r}}, e_{\mathbf{r}} = E_{\mathbf{r}})$$

$$A \textcircled{s} e_{\mathbf{r}} \succ_{\varsigma^{\forall}} \tilde{\mu}x^{\mathbf{s}}.\langle \mu\beta^{\mathbf{r}}.\langle x^{\mathbf{s}} \| A \textcircled{s} \beta^{\mathbf{s}}\rangle \| e_{\mathbf{r}}\rangle \qquad (\nexists E_{\mathbf{r}}, e_{\mathbf{r}} = E_{\mathbf{r}})$$

$$\frac{c \succ c'}{C[c] \to C[c']} \qquad \frac{e \succ e'}{C[e] \to C[e']} \qquad \frac{v \succ v'}{C[v] \to C[v']}$$

Fig. 2. Rewriting theory for multi-discipline, polymorphic sequent calculus.

Parameterized Reduction Theory. The reduction theory, denoted by $\to$ shown in Fig. 2, is the *compatible closure* of the top-level reduction relation $\succ$. Here the metavariable C ranges over any context such that filling the whole with an object of the appropriate sort is well formed. Whereas $\succ$ only applies to the top of some expression, $\to$ can apply *anywhere* inside of it. Further, we use $\twoheadrightarrow$ for the *reflexive, transitive* closure of $\to$. The reduction rules in $\succ$ are given names which we write in subscript. We also use subscripts on the $\to$ rule to denote the restriction to the rules of the same name, for instance $\to_{\beta^{\to}}$ refers to the compatible closure of the relation $\succ_{\beta^{\to}}$. At times we will use multiple subscripts to denote collections of reductions, as in $\succ_{\beta^{\to},\beta^{\forall}}$ for the union of $\succ_{\beta^{\to}}$ and $\succ_{\beta^{\forall}}$. When a relation such as $\succ$ or $\to$ is used without a subscript it refers to the union over all of the rules.

The reduction theory is parameterized by a set of specific discipline symbols equipped with an associated subset of terms called *values* and co-terms called

co-values (denoted by $V_{\mathbf{s}}$ and $E_{\mathbf{s}}$, respectively, for each discipline symbol $\mathbf{s}$). As with (co-)terms, we use the plain metavariables V and E to refer to the union of values and co-values for every $\mathbf{s}$. For example, we write $\langle \mu\boldsymbol{\alpha}.c \parallel E\rangle \to_\mu c\{E/\boldsymbol{\alpha}\}$ and by the syntactic requirement that the two sides of a command agree on a discipline, it must be that the disciplines of E and $\boldsymbol{\alpha}$ match. Disciplines are not just restrictive but also enabling in the case of the ς rules (originally due to Wadler [28]) that lift unevaluated components out of call-stacks to be computed, so there is no "largest" reduction theory that subsumes all others.

$$
\begin{aligned}
V_{\mathbf{u}} &::= v_{\mathbf{u}} & E_{\mathbf{u}} &::= e_{\mathbf{u}} \\
V_{\mathbf{v}} &::= x^{\mathbf{v}} \mid \mu(\boldsymbol{x} \textcircled{v} \boldsymbol{\alpha}).c \mid \mu(\boldsymbol{a} \textcircled{v} \boldsymbol{\alpha}).c & E_{\mathbf{v}} &::= e_{\mathbf{v}} \\
V_{\mathbf{n}} &::= v_{\mathbf{n}} & E_{\mathbf{n}} &::= \alpha^{\mathbf{n}} \mid V \textcircled{n} E \mid A \textcircled{n} E
\end{aligned}
$$

$$
\begin{aligned}
V_{\mathbf{lv}} &::= x^{\mathbf{lv}} \mid \mu(\boldsymbol{x} \textcircled{lv} \boldsymbol{\alpha}).c \mid \mu(\boldsymbol{a} \textcircled{lv} \boldsymbol{\alpha}).c \\
E_{\mathbf{lv}} &::= \alpha^{\mathbf{lv}} \mid \tilde{\mu}x^{\mathbf{lv}}.D[\langle x^{\mathbf{lv}} \| E_{\mathbf{lv}}\rangle] \mid V \textcircled{lv} E \mid A \textcircled{lv} E \\
V_{\mathbf{ln}} &::= x^{\mathbf{ln}} \mid \mu(\boldsymbol{x} \textcircled{ln} \boldsymbol{\alpha}).c \mid \mu(\boldsymbol{a} \textcircled{ln} \boldsymbol{\alpha}).c \mid \mu\alpha^{\mathbf{ln}}.D[\langle V_{\mathbf{ln}} \| \alpha^{\mathbf{ln}}\rangle] \\
E_{\mathbf{ln}} &::= \alpha^{\mathbf{ln}} \mid V \textcircled{ln} E \mid A \textcircled{ln} E \\
D &::= \square \mid \langle v_{\mathbf{lv}} \| \tilde{\mu}y^{\mathbf{lv}}.D\rangle \mid \langle \mu\alpha^{\mathbf{ln}}.D \| e_{\mathbf{ln}}\rangle
\end{aligned}
$$

Fig. 3. (Co-)values in by-name ($\mathbf{n}$), -value ($\mathbf{v}$), -need ($\mathbf{lv}$), -co-need ($\mathbf{ln}$) and $\mathbf{u}$.

Values and Co-values. We can now give interpretations of some specific discipline symbols: the call-by-value ($\mathbf{v}$), -name ($\mathbf{n}$), -need ($\mathbf{lv}$ for "lazy value" [1]), -co-need ($\mathbf{ln}$ for "lazy name") and non-deterministic ($\mathbf{u}$) disciplines are defined by the values and co-values in Fig. 3.

$$
\frac{}{\Gamma, \boldsymbol{x} : A \vdash_\Theta \boldsymbol{x} : A \mid \Delta}\ Var
\qquad
\frac{}{\Gamma \mid \boldsymbol{\alpha} : A \vdash_\Theta \boldsymbol{\alpha} : A, \Delta}\ Co\text{-}Var
$$

$$
\frac{c : (\Gamma \vdash_\Theta \boldsymbol{\alpha} : A, \Delta)}{\Gamma \vdash_\Theta \mu\boldsymbol{\alpha}.c : A \mid \Delta}\ Act
\qquad
\frac{c : (\Gamma, \boldsymbol{x} : A \vdash_\Theta \Delta)}{\Gamma \mid \tilde{\mu}\boldsymbol{x}.c : A \vdash_\Theta \Delta}\ Co\text{-}Act
$$

$$
\frac{\Gamma \vdash_\Theta v : A \mid \Delta \quad \Gamma \mid e : A \vdash_\Theta \Delta}{\langle v \| e\rangle : (\Gamma \vdash_\Theta \Delta)}\ Cut
$$

$$
\frac{\Gamma \vdash_\Theta v : A \mid \Delta \quad \Gamma \mid e : B \vdash_\Theta \Delta}{\Gamma \mid v \textcircled{s} e : A \xrightarrow{\mathbf{s}} B \vdash_\Theta \Delta}\ {\to}L
\qquad
\frac{c : (\Gamma, \boldsymbol{x} : A \vdash_\Theta \boldsymbol{\alpha} : B, \Delta)}{\Gamma \vdash_\Theta \mu(\boldsymbol{x} \textcircled{s} \boldsymbol{\alpha}).c : A \xrightarrow{\mathbf{s}} B \mid \Delta}\ {\to}R
$$

$$
\frac{\Gamma \mid e : B\{A_{\mathbf{t}}/a^{\mathbf{t}}\} \vdash_\Theta \Delta}{\Gamma \mid A_{\mathbf{t}} \textcircled{s} e : \forall^{\mathbf{s}} a^{\mathbf{t}}.B \vdash_\Theta \Delta}\ \forall L
\qquad
\frac{c : (\Gamma \vdash_{\Theta, \boldsymbol{a}} \boldsymbol{\alpha} : B, \Delta)}{\Gamma \vdash_\Theta \mu(\boldsymbol{a} \textcircled{s} \boldsymbol{\alpha}).c : \forall^{\mathbf{s}} \boldsymbol{a}.B \mid \Delta}\ \forall R
$$

Fig. 4. Type system for the multi-discipline, polymorphic sequent calculus.

Typing. As a generalization of polarity, types belong to one of several *kinds*, each associated with a discipline. The kind of a type is specified by its top

constructor, for example $A \overset{\mathbf{v}}{\to} B$ and $A \overset{\mathbf{lv}}{\to} B$ are types of call-by-value and call-by-need, respectively. Type variables range over a specific kind denoting the discipline of (co-)terms they specify, and the polymorphic quantifier $\forall^{\mathbf{s}}$ must choose a specific kind of type to abstract over.

The typing rules for the calculus are given in Fig. 4. There are some criteria for when sequents are well formed: (1) identifiers (a, x, α) in Θ, Γ, and Δ are all unique, (2) the disciplines of (co-)variables must match that of their type, as in $x^{\mathbf{s}} : A_{\mathbf{s}}$ and $\alpha^{\mathbf{s}} : A_{\mathbf{s}}$, and (3) in the sequent $\Gamma \vdash_{\Theta} v : A \mid \Delta$, all the free type variables of Γ, Δ, v, and A are included in Θ, and similarly for $\Gamma \mid e : A \vdash_{\Theta} \Delta$ and $c : (\Gamma \vdash_{\Theta} \Delta)$. Only derivations where all sequents are well formed are considered proofs. Note that this imposes the standard criteria on the right $\forall$ rule that the abstracted type variable in the premise is not free in the conclusion. Well-formedness also ensures that in the cut and the left rule for $\forall$, the free variables of the cut and instantiated type are contained in Θ.

Admissible Disciplines. Our proof of strong normalization is parameterized by a collection of discipline symbols and their interpretation. However, there are two important properties on disciplines needed for our proof.

Definition 1. *A discipline is* stable *exactly when (co-)values are closed under reduction and substitution,* focalizing *exactly when* at least *all (1) variables,* $\mu(\boldsymbol{x} \; ⓢ \; \boldsymbol{\alpha}).c$, *and* $\mu(\boldsymbol{a} \; ⓢ \; \boldsymbol{\alpha}).c$ *are values, and (2) co-variables,* $V \; ⓢ \; E$, *and* $A \; ⓢ \; E$, *are co-values, and* admissible *exactly when it is stable and focalizing.*

Property 1. The **n**, **v**, **lv**, **ln**, and **u** disciplines are collectively admissible.

Our proof of strong normalization works uniformly for any collection of admissible disciplines. As we present the proof in the next section we assume some admissible disciplines have been chosen, which could include any combination of the five disciplines presented above, or some other admissible disciplines of interest.

5 Strong Normalization

While some properties, like type safety, are straightforward enough to prove directly [29], other properties, like strong normalization, resist a direct approach. The problem with proving strong normalization is that just inducting over syntax or typing derivations is far too weak. Instead, the standard practice uses a more indirect approach based on the idea behind Tait's method [25] and reducibility candidates [8]: set up an interpretation for types that serves as a way-point between syntax and safety. The interpretation for a type should encompass all programs of that type (adequacy) and also fit inside the intended candidate property (safety). When interpreting types, the definition is usually designed with safety in mind: interpretations contain only safe programs by construction, but their adequacy needs to be justified. Instead, we will orient ourselves the other way in the style of symmetric candidates [2], where the interpretations for types are designed with adequacy in mind: interpretations contain all the

necessary well-typed programs by construction, but their safety needs to be justified. But that means we need to consider things which are not yet known to be safe, and so are not a candidate interpretation for any type. Therefore, we work in the larger and more lax domain of *pre-types* which encompasses all possible candidates but does not impose the necessary safety conditions.

Pre-types. In the biorthogonality family of methods [7,13,21], a type has a two-sided interpretation described by both a set of terms and a set of co-terms. Intuitively, a model of a type describes some desired behavior of programs (like an algorithm, or specification), where the term side can be seen as a collection of implementations and the co-term side can be seen as set of test operations. By analogy, *orthogonality* is an operation that evaluates implementations (terms) with operations (co-terms). On the one hand, orthogonality selects only those implementations that pass a comprehensive set of tests, and on the other hand, orthogonality also selects only those test that pass the reference implementation(s). The biorthogonal interpretation of types then is safe by construction, where the co-terms (tests) of a type are exactly everything orthogonal to (here, forming a strongly normalizing command with) any term (implementation) of the type, and vice versa. Since orthogonality can always complete one half from the other, only one side is necessary.

However, the method we use here cannot rely on such luxuries. While constructing the interpretation of types, we will have to consider incremental steps which may include extra (co-)terms that create unsafe interactions and exclude necessary (co-)terms that would be safe. Therefore, the new insight is to work in a domain where terms and co-terms are grouped *together* as a single unit, and which includes many *pre-types* that are not candidate interpretations for types.

Definition 2. *A* pre-type $\mathcal{A}$ *(of discipline* $\mathbf{r}$*) is a pair* (A_v, A_e) *where* A_v *is a set of strongly normalizing* $\mathbf{r}$*-terms and* A_e *is a set of strongly normalizing* $\mathbf{r}$*-co-terms.*

We use ordinary set membership to refer to the underlying sets: given $\mathcal{A} = (A_v, A_e)$, we write $v \in \mathcal{A}$ for $v \in A_v$ and $e \in \mathcal{A}$ for $e \in A_e$. We write $\mathcal{SN}_{\mathbf{r}}$ for the pre-type containing all strongly normalizing (co-)terms of discipline $\mathbf{r}$ and $⫫$ for the set of all strongly normalizing commands.

We can compare pre-types like we do with sets. But because they are built with two opposite sets, there are two different methods of comparison. The first comparison is *containment* which just checks that the (co-)terms of one pre-type also belong to the other. The second comparison corresponds instead to behavioral *sub-typing* [16]: A is a sub-type of B if every program fragment of A can be used in every context of B. Intuitively, the subsumption of sub-typing sends every producer of As (*i.e.,* terms) to B and dually sends every consumer of Bs (*i.e.,* co-terms) to A. We can also combine pre-types with unions and intersections that go along with these two comparisons: for containment this just means the union and intersection, respectively, of the sets underlying pre-types, but for sub-typing this corresponds to the intuition behind union and intersection types in programming languages.

Definition 3. *Let $\mathcal{A} = (A_v, A_e)$ and $\mathcal{B} = (B_v, B_e)$ be pre-types. $\mathcal{A}$ is* contained *in $\mathcal{B}$, written $\mathcal{A} \sqsubseteq \mathcal{B}$, and $\mathcal{A}$ is a* sub-type *of $\mathcal{B}$, written $\mathcal{A} \leq \mathcal{B}$, as follows:*

$$\mathcal{A} \sqsubseteq \mathcal{B} \textit{ iff } A_v \subseteq B_v \textit{ and } A_e \subseteq B_e \qquad \mathcal{A} \leq \mathcal{B} \textit{ iff } A_v \subseteq B_v \textit{ and } A_e \supseteq B_e$$

The union and intersection for containment ($\sqcup, \sqcap$) and sub-typing ($\vee, \wedge$) are:

$$\mathcal{A} \sqcup \mathcal{B} = (A_v \cup B_v, A_e \cup B_e) \qquad \mathcal{A} \vee \mathcal{B} = (A_v \cup B_v, A_e \cap B_e)$$
$$\mathcal{A} \sqcap \mathcal{B} = (A_v \cap B_v, A_e \cap B_e) \qquad \mathcal{A} \wedge \mathcal{B} = (A_v \cap B_v, A_e \cup B_e)$$

Orthogonality. The orthogonality operation on pre-types, $\mathcal{A}^{\perp}$, uses one pre-type to generate another one containing everything it can safely interact with and nothing more. Together, orthogonality and containment capture the notion of safety in terms of pre-types: $\mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$ means $\langle v \| e \rangle \in \perp\!\!\!\perp$ for all $v, e \in \mathcal{A}$.

Definition 4. *The orthogonal of any pre-type $\mathcal{A}$ of $\mathbf{r}$, written $\mathcal{A}^{\perp}$, is:*

$$v_{\mathbf{r}} \in \mathcal{A}^{\perp} \iff \forall e_{\mathbf{r}} \in \mathcal{A}. \langle v_{\mathbf{r}} \| e_{\mathbf{r}} \rangle \in \perp\!\!\!\perp \quad e_{\mathbf{r}} \in \mathcal{A}^{\perp} \iff \forall v_{\mathbf{r}} \in \mathcal{A}. \langle v_{\mathbf{r}} \| e_{\mathbf{r}} \rangle \in \perp\!\!\!\perp$$

A pre-type $\mathcal{A}$ of $\mathbf{r}$ is safe *exactly when $\mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$.*

Although we have generalized the notion of orthogonality to pre-types, it still exhibits the properties which mimic negation in intuitionistic logic.

Property 2. Contrapositive: If $\mathcal{A} \sqsubseteq \mathcal{B}$ then $\mathcal{B}^{\perp} \sqsubseteq \mathcal{A}^{\perp}$. *Double orthogonal introduction*: $\mathcal{A} \sqsubseteq \mathcal{A}^{\perp\perp}$. *Triple orthogonal elimination*: $\mathcal{A}^{\perp\perp\perp} = \mathcal{A}^{\perp}$.

However, because pre-types also come with another notion of comparison, we get an additional *new* property of orthogonality that follows from sub-typing.

Property 3. Monotonicity: If $\mathcal{A} \leq \mathcal{B}$ then $\mathcal{A}^{\perp} \leq \mathcal{B}^{\perp}$.

This fact is key to our entire endeavor: the monotonicity of orthogonality (and similar operations) with respect to sub-typing guarantees that there are always fixed points of orthogonality. This is the fact that powers the fixed-point construction of symmetric candidates [2] that we generalize by rephrasing the construction in terms of a two-sided model of sub-typing.

Top Reduction. Another standard part of a strong normalization proof is to identify a subset of reductions that are important to check for the purpose of normalization. Usually in the λ-calculus, these important reductions are the standard reductions that make up an operational semantics. But since we are working in the sequent calculus, we already have a notion of "main" reduction that is immediately apparent in the syntax: the reductions that occur at the "top" of a command. We define *top reduction* $\rightsquigarrow$ on commands as:

$$\frac{c \succ_{\beta^{\rightarrow}, \beta^{\forall}} c'}{c \rightsquigarrow_0 c'} \qquad \frac{c \succ_{\mu} c'}{c \rightsquigarrow_+ c'} \qquad \frac{c \succ_{\tilde{\mu}} c'}{c \rightsquigarrow_- c'} \qquad \frac{e_{\mathbf{r}} \succ_{\varsigma^{\rightarrow}, \varsigma^{\forall}} e'_{\mathbf{r}}}{\langle V_{\mathbf{r}} \| e_{\mathbf{r}} \rangle \rightsquigarrow_- \langle V_{\mathbf{r}} \| e'_{\mathbf{r}} \rangle} \qquad \frac{c \rightsquigarrow_{+,0,-} c'}{c \rightsquigarrow c'}$$

Note that top reductions are distinguished based on a "charge:" the positive $\rightsquigarrow_+$ let the term of a command take control of computation, the negative $\rightsquigarrow_-$ let the co-term take control, and the neutral $\rightsquigarrow_0$ require that both the term and co-term cooperate to proceed. The purpose of this distinction is to help tame the potential for non-determinism: notice that both $\rightsquigarrow_{+,0}$ and $\rightsquigarrow_{-,0}$ are deterministic for *all* disciplines, but $\rightsquigarrow_{+,-}$ *may not* be depending on the discipline. We need to pay attention to non-determinism because it breaks the expected expansion property used in strong normalization proofs [9]. Normally, top expansion says that if $\langle v \| e \rangle \rightsquigarrow c$ and v, e, and c are all strongly normalizing then so is $\langle v \| e \rangle$. However, this might not work if there is another top reduction $\langle v \| e \rangle \rightsquigarrow c'$ where c' loops forever. So generalizing top expansion to accommodate non-determinism must quantify over *all* possible top reductions; even after some other internal reductions have happened. Within a pre-type $\mathcal{A}$, non-deterministic top expansion assumes $\mathcal{A}$ is closed under reduction—if $v, e \in \mathcal{A}$ and $v \twoheadrightarrow v'$ and $e \twoheadrightarrow e'$ then $v', e' \in \mathcal{A}$—and that every possible top reduction of $\mathcal{A}$ commands leads to a strongly normalizing command. If so, every $\mathcal{A}$ command *is* strongly normalizing.

Lemma 1 (Nondeterministic Top Expansion). *If $\mathcal{A}$ is closed under reduction and for all $v, e \in \mathcal{A}$ and c, $\langle v \| e \rangle \rightsquigarrow c$ implies $c \in \perp\!\!\!\perp$, then $\mathcal{A}$ is safe.*

So we have a top expansion property for the general non-deterministic case, but what about when we are dealing with (co-)terms from a deterministic discipline like $\mathbf{v}$ or $\mathbf{n}$? We can identify a pre-type of deterministically-normalizing (co-)terms of $\mathbf{r}$ ($\mathcal{DN}_\mathbf{r}$) where *all* their possible top reductions either land in $\perp\!\!\!\perp$ or not after any number of other reductions have occurred, which is defined as:

$$v_\mathbf{r} \in \mathcal{DN}_\mathbf{r} \iff \forall e_\mathbf{r} \in \mathcal{SN}_\mathbf{r}.(v_\mathbf{r} \twoheadrightarrow v'_\mathbf{r} \wedge \langle v'_\mathbf{r} \| e_\mathbf{r} \rangle \rightsquigarrow c \wedge \langle v'_\mathbf{r} \| e_\mathbf{r} \rangle \rightsquigarrow c') \Rightarrow (c \in \perp\!\!\!\perp \Leftrightarrow c' \in \perp\!\!\!\perp)$$
$$e_\mathbf{r} \in \mathcal{DN}_\mathbf{r} \iff \forall v_\mathbf{r} \in \mathcal{SN}_\mathbf{r}.(e_\mathbf{r} \twoheadrightarrow e'_\mathbf{r} \wedge \langle v_\mathbf{r} \| e'_\mathbf{r} \rangle \rightsquigarrow c \wedge \langle v_\mathbf{r} \| e'_\mathbf{r} \rangle \rightsquigarrow c') \Rightarrow (c \in \perp\!\!\!\perp \Leftrightarrow c' \in \perp\!\!\!\perp)$$

As shorthand, we write $\mathcal{A}^d$ to mean $\mathcal{A} \sqcap \mathcal{DN}_\mathbf{r}$ for a pre-type $\mathcal{A}$ of $\mathbf{r}$. Now, we get an improved top expansion property for deterministically-normalizing (co-)terms.

Lemma 2 (Deterministic top expansion). *If $\mathbf{r}$ is stable, $v, e \in \mathcal{SN}_\mathbf{r}$, either $v \in \mathcal{DN}_\mathbf{r}$ or $e \in \mathcal{DN}_\mathbf{r}$, and $\langle v \| e \rangle \rightsquigarrow c \in \perp\!\!\!\perp$ then $\langle v \| e \rangle \in \perp\!\!\!\perp$.*

Deterministic top expansion relies on commutation between top and non-top reductions based on the stability of $\mathbf{r}$. Note that for any discipline $\mathbf{r}$ where top reduction is deterministic, it follows that $\mathcal{SN}_\mathbf{r} = \mathcal{DN}_\mathbf{r}$, and so the above deterministic top expansion property holds for *any* term and co-term of $\mathbf{r}$. Since the $\mathbf{n}$, $\mathbf{v}$, $\mathbf{lv}$, and $\mathbf{ln}$ disciplines all meet this criteria, they all enjoy the usual expansion property unlike $\mathbf{u}$.

Reducibility Candidates. The interpretation of a type should be both adequate and safe. Safety, which tells us a type's interpretation contains only good interactions, was already captured by orthogonality ($\mathcal{A}$ is safe when $\mathcal{A} \sqsubseteq \mathcal{A}^\perp$). Adequacy, which tells us a type's interpretation contains all programs dictated

by the typing rules, is a little more involved, however. Certainly, interpretations should include everything that interacts well with the type $(\mathcal{A}^{\perp} \sqsubseteq \mathcal{A})$, but this is not enough. We need to be able to show type membership looking at a single top reduction, but reduction isn't in general deterministic, so we must explicitly require that a (co-)term that interacts well with $\mathcal{A}$ after it causes one top reduction is also in $\mathcal{A}$. This extra condition only tests the (co-)values of $\mathcal{A}$: $\langle v_{\mathbf{r}} \| e_{\mathbf{r}} \rangle \rightsquigarrow_{+} c$ only when $e_{\mathbf{r}}$ is a co-value of $\mathbf{r}$ and $\langle v_{\mathbf{r}} \| e_{\mathbf{r}} \rangle \rightsquigarrow_{-} c$ only when $v_{\mathbf{r}}$ is a value of $\mathbf{r}$.

Definition 5. *The* saturation *of a pre-type $\mathcal{A}$ of $\mathbf{r}$ is defined as:*

$$v_{\mathbf{r}} \in \mathcal{A}^{s} \iff \forall E_{\mathbf{r}} \in \mathcal{A}. \langle v_{\mathbf{r}} \| E_{\mathbf{r}} \rangle \rightsquigarrow_{+,0}^{=} c \in \perp\!\!\!\perp \qquad e_{\mathbf{r}} \in \mathcal{A}^{s} \iff \forall V_{\mathbf{r}} \in \mathcal{A}. \langle V_{\mathbf{r}} \| e_{\mathbf{r}} \rangle \rightsquigarrow_{-,0}^{=} c \in \perp\!\!\!\perp$$

where $\rightsquigarrow_{+,0}^{=}$ and $\rightsquigarrow_{-,0}^{=}$ are the reflexive closures of $\rightsquigarrow_{+,0}$ and $\rightsquigarrow_{-,0}$, respectively. A pre-type $\mathcal{A}$ of $\mathbf{r}$ is adequate *exactly when $\mathcal{A}^{s} \sqsubseteq \mathcal{A}$.*

Now that we know how to phrase safety in terms of orthogonality and adequacy in terms of saturation, we can say that *reducibility candidates*, which are the potential interpretations of types, are pre-types that lie between their own saturation and orthogonal.

Definition 6. *A* reducibility candidate *$\mathcal{A}$ (* of $\mathbf{r}$*) is a safe and adequate pre-type $\mathcal{A}$ of $\mathbf{r}$ (*i.e., $\mathcal{A}^{s} \sqsubseteq \mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$*). $CR_{\mathbf{r}}$ is the set of all reducibility candidates of $\mathbf{r}$.*

In practice, the $\mathcal{A}^{\perp}$ upper-bound is used to justify the cut rule for forming commands, and the $\mathcal{A}^{s}$ lower-bound is used to justify the left and right rules for activation, implication, and universal quantification. Also, the $\mathcal{A}^{s}$ lower-bound serves a second purpose by ensuring that reducibility candidates are all inhabited by (co-)variables, which will be needed to show that typing implies strong normalization even for open commands and (co-)terms.

As it turns out, there is an equivalent way of identifying reducibility candidates of admissible disciplines: they are all fixed points of saturation.

Lemma 3 (Reducibility fixed-point). *For any pre-type $\mathcal{A}$ of an admissible discipline $\mathbf{r}$, $\mathcal{A}$ is a reducibility candidate of $\mathbf{r}$ if and only if $\mathcal{A} = \mathcal{A}^{s}$.*

Reducibility candidates *are* fixed points of saturation because $\mathcal{A}^{\perp} \sqsubseteq \mathcal{A}^{s}$ for any $\mathcal{A}$, and the reverse follows from the focalization of $\mathbf{r}$ because the participants in neutral β-reductions—abstractions and call stacks—are (co-)values that can be tested by saturation. The equivalence between candidates and fixed points gives us a general-purpose construction method for candidates of *any* admissible discipline by solving recursive pre-type equations.

Fixed-Point Solutions. The fixed-point construction of types is powered by the pervasive monotonicity properties of sub-typing between pre-types. Monotonicity isn't limited to just orthogonality; other operations, like saturation and containment-union with a constant pre-type, are also monotonic: for any $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ of $\mathbf{r}$, if $\mathcal{A} \leq \mathcal{B}$ then $\mathcal{A}^{s} \leq \mathcal{B}^{s}$ and $\mathcal{A} \sqcup \mathcal{C} \leq \mathcal{A} \sqcup \mathcal{C}$. Therefore, if we describe the

essence of a type with some pre-type $\mathcal{C}$, we can build a fully-saturated pre-type around it by finding a solution to the equation $\mathcal{A} = \mathcal{C} \sqcup \mathcal{A}^s$. Combined with the fact that sub-typing (and containment) forms a lattice on pre-types, the Knaster-Tarksi fixed point theorem ensures that this equation has a fixed point, giving us the basis of a function for generating saturated pre-types.

Lemma 4 (Fixed-point construction). *For every discipline* $\mathbf{r}$*, there is a function* $\mathcal{F}_{\mathbf{r}}(-)$ *such that for any pre-type* $\mathcal{C}$ *of* $\mathbf{r}$*,* $\mathcal{F}_{\mathbf{r}}(\mathcal{C}) = \mathcal{C} \sqcup \mathcal{F}_{\mathbf{r}}(\mathcal{C})^s$*.*

The Knaster-Tarski fixed point theorem, however, does not ensure that there is a *unique* fixed point satisfying the equation. Therefore, the $\mathcal{F}_{\mathbf{r}}(-)$ operations must somehow pick which among the possible solutions is *the* result. Two readily available options are the largest or smallest such fixed points with respect to sub-typing, but note that neither one is "more principled" than the other: the largest one has the most terms but fewest co-terms, and the smallest one has the fewest terms but most co-terms. Either one will work for demonstrating strong normalization, however, as long as we are consistent. Moreover, we will prove in the next section (Lemma 7) that for deterministic $\mathbf{r}$ the solutions will be unique.

So now we know how to build a saturated extension of any pre-type $\mathcal{C}$ of $\mathbf{r}$ that satisfies one of the conditions for being a reducibility candidate by definition: $\mathcal{F}_{\mathbf{r}}(\mathcal{C})^s \sqsubseteq \mathcal{F}_{\mathbf{r}}(\mathcal{C})$. But we still need to make sure that this extension is safe: we must show that $\mathcal{F}_{\mathbf{r}}(\mathcal{C}) \sqsubseteq \mathcal{F}_{\mathbf{r}}(\mathcal{C})^{\perp}$. It turns out that the safety condition of reducibility candidates follows when $\mathcal{C}$ is a pre-type consisting of only deterministically-normalizing (co-)values that only form strongly-normalizing commands, because then the result of $\mathcal{F}_{\mathbf{r}}(\mathcal{C})$ is itself a fixed point of saturation.

Lemma 5 (Fixed-point validity). *If* $\mathcal{C} \sqsubseteq \mathcal{C}^{\perp dv}$ *then* $\mathcal{F}_{\mathbf{r}}(\mathcal{C}) = \mathcal{F}_{\mathbf{r}}(\mathcal{C})^s$*.*

Where we write $\mathcal{V}_{\mathbf{r}}$ for the pre-type of strongly normalizing (co-)values of discipline $\mathbf{r}$, and use the shorthand $\mathcal{A}^v = \mathcal{A} \sqcap \mathcal{V}_{\mathbf{r}}$ for pre-types $\mathcal{A}$ in $\mathbf{r}$.

Interpretations of Types. With a uniform method for generating reducibility candidates in hand, we can now construct the candidates for particular types. Both implication and universal quantification are *negative* types defined by their observations—call stacks—so their interpretation starts with the negative construction of a pre-type that selects terms compatible with some co-terms: for a set of strongly-normalizing $\mathbf{r}$-co-terms O, $Neg(O)$ is the following pre-type of $\mathbf{r}$:

$$v_{\mathbf{r}} \in Neg(O) \iff \forall E_{\mathbf{r}} \in O. \langle v_{\mathbf{r}} \| E_{\mathbf{r}} \rangle \in \perp\!\!\!\perp \qquad e_{\mathbf{r}} \in Neg(O) \iff e_{\mathbf{r}} \in O$$

The above negative construction satisfies the validity criteria for the fixed-point reducibility candidates from Lemma 5 ($\mathcal{C} \sqsubseteq \mathcal{C}^{\perp dv}$) by keeping only its deterministically-normalizing (co-)values and closing it under orthogonality.

Lemma 6. *For any set* O *of deterministically-normalizing* $\mathbf{r}$*-co-values,* $Neg(O)^{dv} \sqsubseteq Neg(O)^{dv\perp dv} = Neg(O)^{dv\perp dv\perp dv}$

We now have a negative interpretation for the specific type constructors:

- For all $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A} \xrightarrow{\mathbf{r}} \mathcal{B} \triangleq \mathcal{F}_{\mathbf{r}}(Neg(\{V \,\textcircled{r}\, E \mid V \in \mathcal{A}, E \in \mathcal{B}\})^{dv\perp dv}) \in CR_{\mathbf{r}}$.
- For all $K \subseteq CR_{\mathbf{t}}$, $\forall^{\mathbf{r}}\mathbf{t}.K \triangleq \mathcal{F}_{\mathbf{r}}(Neg(\{A_{\mathbf{t}} \,\textcircled{r}\, E \mid \mathcal{B} \in K, E \in \mathcal{B}\})^{dv\perp dv}) \in CR_{\mathbf{r}}$.

Adequacy. The final step is to give an interpretation for syntactic types, environments, and sequents as reducibility candidates, substitutions, and propositions, respectively, where we write CR for $\bigcup_{\mathbf{r}} CR_{\mathbf{r}}$:

$$[\![a^{\mathbf{r}}]\!]\phi \triangleq \phi(a) \quad [\![A \xrightarrow{\mathbf{r}} B]\!]\phi \triangleq [\![A]\!]\phi \xrightarrow{\mathbf{r}} [\![B]\!]\phi \quad [\![\forall^{\mathbf{r}} a^{\mathbf{t}}.B]\!]\phi \triangleq \forall^{\mathbf{r}}\mathbf{t}.\{[\![B]\!](\phi, \mathcal{A}/a^{\mathbf{t}}) \mid \mathcal{A} \in CR_{\mathbf{t}}\}$$

$$[\![\Theta]\!] \triangleq \{\phi \mid \forall a^{\mathbf{r}} \in \Theta.\phi(a) \in CR_{\mathbf{r}}\}$$

$$[\![\Gamma \vdash_{\Theta} \Delta]\!]\phi \triangleq \{\rho \mid \forall a^{\mathbf{r}} \in \Theta.a\{\rho\} \in Type_{\mathbf{r}} \wedge \forall \boldsymbol{x}{:}A \in \Gamma.\boldsymbol{x}\{\rho\} \in [\![A]\!]\phi \wedge \forall \boldsymbol{\alpha}{:}A \in \Delta.\boldsymbol{\alpha}\{\rho\} \in [\![A]\!]\phi\}$$

$$c : (\Gamma \vDash_{\Theta} \Delta) \triangleq \forall \phi \in [\![\Theta]\!], \rho \in [\![\Gamma \vdash_{\Theta} \Delta]\!]\phi.c\{\rho\} \in \perp\!\!\!\perp$$

$$\Gamma \vDash_{\Theta} v : A \mid \Delta \triangleq \forall \phi \in [\![\Theta]\!], \rho \in [\![\Gamma \vdash_{\Theta} \Delta]\!]\phi.[\![A]\!]\phi \in CR \wedge v\{\rho\} \in [\![A]\!]\phi$$

$$\Gamma \mid e : A \vDash_{\Theta} \Delta \triangleq \forall \phi \in [\![\Theta]\!], \rho \in [\![\Gamma \vdash_{\Theta} \Delta]\!]\phi.[\![A]\!]\phi \in CR \wedge e\{\rho\} \in [\![A]\!]\phi$$

Typing derivations are adequate with respect to the interpretation of their conclusion for any admissible disciplines, which in turn gives us strong normalization.

Theorem 1 (Adequacy). *(1) $c : (\Gamma \vdash_{\Theta} \Delta)$ implies $c : (\Gamma \vDash_{\Theta} \Delta)$, (2) $\Gamma \vdash_{\Theta} v : A \mid \Delta$ implies $\Gamma \vDash_{\Theta} v : A \mid \Delta$, and (3) $\Gamma \mid e : A \vdash_{\Theta} \Delta$ implies $\Gamma \mid e : A \vDash_{\Theta} \Delta$.*

Adequacy follows by induction on the typing derivation. Note that the requirement that disciplines are focalizing is used to justify the left and right rules of functions and polymorphism so that abstractions and call stacks end up in the meaning of those types. This also ensures that (co-)variables are (co-)values (*resp.*) that inhabit every reducibility candidate, so that every environment has a suitable substitution used to extract strong normalization for reduction of open commands, terms, and co-terms.

Corollary 1 (Strong normalization). *Typed commands, terms, and co-terms are strongly normalizing.*

6 Biorthogonals are Fixed Points

The candidate-based approach to strong normalization—tracing back to Tait [25] and Girard [6] and fitting in the general area of logical relations [27] and realizability [11]—easily accommodates impredicative polymorphism by outlining the candidate meanings of types before defining any particular type. Tait's original method doesn't work for us because we need types to classify co-terms in addition to terms. The use of orthogonality for modeling types appears in multiple places, including Girard's [7] linear logic, Krivine's [13] classical realizability, and Pitts' [21] $\top\top$-closed relations, and can prove strong normalization for certain disciplines. For call-by-name we could start by defining types via their observations (so for functions, valid call stacks), the set of terms of that type as anything orthogonal to these observations, and, finally, the set of co-terms of that type as the double orthogonal of the defining observations. The dual approach, starting with the constructions of values, works for call-by-value.

Munch-Maccagnoni [17] identified a key feature of the orthogonal construction of types: all call-by-value and -name types are generated by their values and co-values, respectively. That is, the meaning of a type *is* the orthogonal of its (co-)values; in our notation, $\mathcal{A} = \mathcal{A}^{v\perp}$. As it turns out, these are exactly the reducibility candidates produced by our fixed-point framework for well-behaved disciplines that induce enough determinism. In the general case, the inherent non-determinism of disciplines like **u** allows for many different and incompatible candidate meanings for a particular type [14], but for disciplines like **v**, **n**, **lv**, and **ln** that eliminate the fundamental non-deterministic choice, there can only be one meaning for each type and it must be the fixed point of $-^{v\perp}$.

Lemma 7. *For any admissible discipline* $\mathbf{r}$ *where* $\mathcal{SN}_{\mathbf{r}} = \mathcal{DN}_{\mathbf{r}}$ *and pre-type* $\mathcal{A}$ *of* $\mathbf{r}$*,* $\mathcal{A}$ *is the* unique *reducibility candidate containing* $\mathcal{A}^v$ *if and only if* $\mathcal{A} = \mathcal{A}^{v\perp}$*.*

This extra uniqueness property of candidates provided by determinism gives us a more direct method of building them in a finite number of steps, as opposed to using the existence of solutions to recursive equations. In particular, note that there is a *positive* construction of pre-types, dual to $Neg(-)$ from Sect. 5, which uses some set of terms C to generate all compatible co-terms:

$$v \in Pos(C) \iff v \in C \qquad e \in Pos(C) \iff \forall v \in C. \langle v \| e \rangle \in \perp\!\!\!\perp$$

Both the positive and negative construction of pre-types can be used to directly construct reducibility candidates of any deterministic discipline (as in Lemma 7). In the special cases of call-by-value and call-by-name there is an even simpler construction because they trivialize the co-value- and value-restriction, respectively.

Theorem 2. *Let* $\mathbf{r}$ *be any admissible discipline with deterministic top reduction (including* $\mathbf{v}$*,* $\mathbf{n}$*,* $\mathbf{lv}$*, and* $\mathbf{ln}$*, among others),* C *be a set of* $\mathbf{r}$*-values, and* O *be a set of* $\mathbf{r}$*-co-values. Both* $Pos(C)^{v\perp v\perp}$ *and* $Neg(O)^{v\perp v\perp}$ *are reducibility candidates of* $\mathbf{r}$*. Furthermore,* $Pos(C)^{\perp} \in CR_{\mathbf{v}}$ *if* $\mathbf{r} = \mathbf{v}$ *and* $Neg(O)^{\perp} \in CR_{\mathbf{n}}$ *if* $\mathbf{r} = \mathbf{n}$*.*

The finitely-constructed candidates $Neg(O)^{\perp} \in CR_{\mathbf{n}}$ and $Pos(C)^{\perp} \in CR_{\mathbf{v}}$ are exactly the usual biorthogonal meanings of types in call-by-name and call-by-value languages: both $Neg(O)$ and $Pos(C)$ include a built-in orthogonal on one side of the pre-type to get started, and the second orthogonal is a closure operation since any more are redundant ($Neg(O)^{\perp\perp} = Neg(O)^{\perp}$ and $Pos(C)^{\perp\perp} = Pos(C)^{\perp}$). For example, let the set of call-stacks for two pre-types $\mathcal{A}$ and $\mathcal{B}$ be $\mathcal{A} \,❶_{\mathbf{r}}\, \mathcal{B} = \{V \,❶_{\mathbf{r}}\, E \mid V \in \mathcal{A}, E \in \mathcal{B}\}$, so that the interpretation of call-by-name and -value function types *must* be

$$\mathcal{A} \xrightarrow{\mathbf{n}} \mathcal{B} = ((\mathcal{A} \,❶_{\mathbf{n}}\, \mathcal{B})^{\perp}, (\mathcal{A} \,❶_{\mathbf{n}}\, \mathcal{B})^{\perp\perp}) \quad \mathcal{A} \xrightarrow{\mathbf{v}} \mathcal{B} = ((\mathcal{A} \,❶_{\mathbf{v}}\, \mathcal{B})^{\perp v\perp\perp}, (\mathcal{A} \,❶_{\mathbf{v}}\, \mathcal{B})^{\perp v\perp})$$

Theorem 2 also gives the first finite construction of reducibility candidates for call-by-need and its dual, which only differs from the simple biorthogonal mean-

ings by being careful about (co-)values and using one more level of orthogonality to reach a fixed point. For example, lazy function types, where **l** is **lv** or **ln**, *must* be

$$\mathcal{A} \xrightarrow{\mathbf{l}} \mathcal{B} = ((\mathcal{A} \mathbin{❶} \mathcal{B})^{\perp v \perp v \perp}, (\mathcal{A} \mathbin{❶} \mathcal{B})^{\perp v \perp})$$

The uniqueness condition of Lemma 7 removes any other possibilities for call-by-name and -value specifically—$Neg(O)^{\perp} \in CR_{\mathbf{n}}$ and $Pos(C)^{\perp} \in CR_{\mathbf{v}}$ are the *only* candidates containing $Neg(O)^{\perp v}$ and $Pos(C)^{\perp v}$—and similarly for the general-purpose positive and negative candidates. That means the candidates of **n**, **v**, **lv**, and **ln**, and any other deterministic, admissible discipline produced by our general-purpose fixed-point construction must be exactly these, so our framework subsumes the existing discipline-specific biorthogonal methods for (any combinations of) call-by-name and -value.

In comparison with Barbanera and Berardi's symmetric candidates method [2] for the symmetric λ-calculus—a calculus corresponding to **u** since all (co-)terms are substitutable and there are no ς-reductions—there are more differences. The main underlying idea to generate candidates as the fixed point of some saturation operation is the same, as is the definition of candidates as something in between saturation and orthogonality, but the meaning of "saturation" used here is more general. In particular, symmetric candidates defines saturation in terms of the syntax of programs, requiring that (co-)variables and certain μ- and $\tilde{\mu}$-abstractions satisfying some conditions are present. We instead define saturation in terms of the behavior of programs, requiring that they work—either now or in one step—with all relevant (co-)values. When considering only the **u** discipline, the approaches produce identical candidates. However, basing saturation on dynamic structure instead of syntactic structure has two benefits. First, it is straightforward to extend the basic method to accommodate additional language features, like multiple disciplines and focusing via ς-reductions as we have done here, since the meaning of saturation does not have to change: run-time behavior is enough to uniformly describe new features. Second, our definition of saturation is strictly more inclusive than the one of symmetric candidates: *everything* that works *must* be included. The larger saturation is key for Lemma 7, and Theorem 2 and for subsuming the biorthogonal methods in the more general multi-discipline setting: since we know that candidates include all the sensible (co-)terms, there is less room for spurious variations making the final result more precise.

7 Conclusion

We have explored multi-discipline calculi with polymorphism and control, based on the sequent calculus. The sequent calculus setting is good for exploring multi-discipline programming since it provides a clean separation between the different disciplines and allows us to treat them abstractly as an object of study. As our main objective, we established strong normalization by using a model of types

based on both orthogonality and fixed points. Our model is uniform over multiple disciplines, with a generic characterization of which ones are admissible, and strictly generalizes several previous models. This study illustrates the benefits of both the sequent calculus and discipline-agnostic reasoning: we can give a single explanation for several calculi in one fell swoop and without losing anything from the discipline-specific models. Our setting of pre-types already comes with a built-in notion of sub-typing along with the union and intersection of types, it would be interesting to relate these ideas to filter models and the characterization of strong normalization in terms of intersection types. More practically, we would like to relate our formal study of mixing disciplines to the way current languages combine strict and lazy features, with an ultimate aim of improving multi-disciplined programming and compilation.

Acknowledgments. This work is supported by the National Science Foundation under grants CCF-1719158 and CCF-1423617.

References

1. Ariola, Z.M., Herbelin, H., Saurin, A.: Classical call-by-need and duality. In: TLCA 2011 (2011)
2. Barbanera, F., Berardi, S.: A symmetric lambda calculus for "classical" program extraction. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 495–515. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-57887-0_112
3. Curien, P.L., Herbelin, H.: The duality of computation. In: ICFP 2000 (2000)
4. Downen, P., Ariola, Z.M.: The duality of construction. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 249–269. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_14
5. Downen, P., Johnson-Freyd, P., Ariola, Z.M.: Structures for structural recursion. In: ICFP 2015 (2015)
6. Girard, J.Y.: Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur. These d'état, Université de Paris 7 (1972)
7. Girard, J.Y.: Linear logic. Theoret. Comput. Sci. **50**, 1–102 (1987)
8. Girard, J.Y., Taylor, P., Lafont, Y.: Proofs and Types. Cambridge University Press, New York (1989)
9. Graham-Lengrand, S.: Polarities & Focussing: a journey from Realisability to Automated Reasoning. Habilitation thesis, Université Paris-Sud (2014)
10. Herbelin, H., Zimmermann, S.: An operational account of call-by-value minimal and classical λ-calculus in "natural deduction" form. In: TLCA 2009 (2009)
11. Kleene, S.C.: On the interpretation of intuitionistic number theory. J. Symbolic Logic **10**(4), 109–124 (1945)
12. Krivine, J.L.: A call-by-name lambda-calculus machine. High. Order Symbolic Comput. **20**(3), 199–207 (2007)
13. Krivine, J.L.: Realizability in classical logic. In: Interactive models of computation and program behaviour, vol. 27. Société Mathématique de France (2009)
14. Lengrand, S., Miquel, A.: Classical Fω, orthogonality and symmetric candidates. Ann. Pure Appl. Logic **153**(1), 3–20 (2008)
15. Levy, P.B.: Call-By-Push-Value. Ph.D. thesis, University of London, August 2001

16. Liskov, B.: Keynote address-data abstraction and hierarchy. In: OOPSLA 1987 (1987)
17. Munch-Maccagnoni, G.: Focalisation and classical realisability. In: CSL 2009 (2009)
18. Munch-Maccagnoni, G.: Syntax and Models of a non-Associative Composition of Programs and Proofs. Ph.D. thesis, Université Paris Diderot (2013)
19. Peyton Jones, S.: https://www.red-gate.com/simple-talk/opinion/geek-of-the-week/simon-peyton-jones-geek-of-the-week/
20. Peyton Jones, S.L., Launchbury, J.: Unboxed values as first class citizens in a non-strict functional language. In: FPCA, pp. 636–666 (1991)
21. Pitts, A.M.: Parametric polymorphism and operational equivalence. Math. Struct. Comput. Sci. **10**(3), 321–359 (2000)
22. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theoret. Comput. Sci. **1**, 125–159 (1975)
23. Ronchi Della Rocca, S., Paolini, L.: The Parametric λ-Calculus: A Metamodel for Computation. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-10394-4
24. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. In: LFP 1992, pp. 288–298 (1992)
25. Tait, W.W.: Intensional interpretations of functionals of finite type I. J. Symbolic Logic **32**, 198–212 (1967)
26. Turbak, F., Gifford, D., Sheldon, M.A.: Design Concepts in Programming Languages. The MIT Press (2008)
27. Wadler, P.: Theorems for free! In: FPCA 1989 (1989)
28. Wadler, P.: Call-by-value is dual to call-by-name. In: ICFP 2003 (2003)
29. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. **115**(1), 38–94 (1994)
30. Zeilberger, N.: The logical basis of evaluation order and pattern-matching. Ph.D. thesis, Carnegie Mellon University (2009)

Real-Time Rewriting Logic Semantics for Spatial Concurrent Constraint Programming

Sergio Ramírez[1(✉)], Miguel Romero[1(✉)], Camilo Rocha[1(✉)], and Frank Valencia[1,2(✉)]

[1] Department of Electronics and Computer Science, Pontificia Universidad Javeriana, Cali, Colombia
{sergio,miguel.romero,camilo.rocha,fdvalencia}@javerianacali.edu.co
[2] CNRS, LIX École Polytechnique de Paris, Paris, France

Abstract. Process calculi provide a language in which the structure of *terms* represents the structure of processes together with an *operational semantics* to represent computational steps. This paper uses rewriting logic for specifying and analyzing a process calculus for *concurrent constraint programming* (ccp), combining spatial and real-time behavior. In these systems, agents can run processes in different computational spaces (e.g., containers) while subject to real-time requirements (e.g., upper bounds in the execution time of a given operation), which can be specified with both discrete and dense linear time. The real-time rewriting logic semantics is fully executable in Maude with the help of rewriting modulo SMT: partial information (i.e., constraints) in the specification is represented by quantifier-free formulas on the shared variables of the system that are under the control of SMT decision procedures. The approach is used to symbolically analyze existential real-time reachability properties of process calculi in the presence of spatial hierarchies for sharing information and knowledge.

1 Introduction

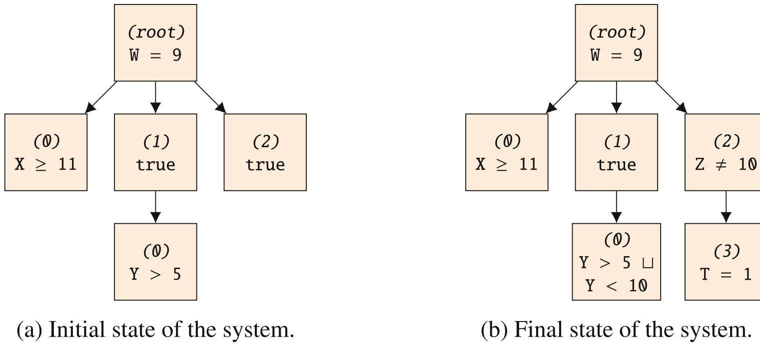
Concurrent constraint programming (ccp). [26] is a well-established process model for concurrency based upon the shared-variables communication model. Its basic intuitions arise mostly from logic; in fact, ccp processes can be interpreted both as concurrent computational entities and logic specifications (e.g., process composition can be seen as parallel execution and as conjunction). In ccp, agents can interact by *posting* (or *telling*) partial information in a medium such as a centralized *store*. Partial information is represented by constraints (e.g., $x > 42$) on the shared variables of the system. The other way in which agents can interact is by *querying* (or *asking*) about partial information entailed by the store. This provides the synchronization mechanism of the model: asking agents are suspended until there is enough information in the store to answer their query. As other mature models of concurrency, ccp has been extended to

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 226–244, 2018.
https://doi.org/10.1007/978-3-319-99840-4_13

capture aspects such as mobility [4,9,12], stochastic behavior [10], and —most prominently— temporal behavior [6,11,16,23,25] for timed and reactive computations, where processes can be constrained also by unit delays and time-out conditions.

However, due to their centralised notion of store, all the previously-mentioned extensions are unsuitable for today's systems where information and processes can be spatially distributed among certain groups of agents. Examples of these systems include agents posting and querying information in the presence of *spatial hierarchies for sharing information and knowledge*, such as friend circles and shared albums in social networks, or shared folders in cloud storage. Recently, the authors of [13] enhanced and generalized the theory of ccp for systems with spatial distribution of information in the novel *spatial concurrent constraint programming* (sccp), where computational hierarchical spaces can be assigned to belong to agents. In sccp, each space may have ccp processes and other (sub) spaces, processes can post and query information in their given space (i.e., locally), and may as well move from one space to another.

As an example, consider the tree-like structures depicted in Fig. 1. They correspond to hierarchical computational spaces of, e.g., virtual containerization (i.e., virtual machines inside other virtual machines). Each one of these spaces is endowed with an agent identifier (either `root` or a natural number) and a local store (i.e., a constraint), and the processes can be executed and spawned concurrently inside any space, with the potential to traverse the structure, querying and posting information locally, and even creating new spaces. The sccp calculus enables the formal modeling of such scenarios and of transitions that can lead from an initial system state (e.g., Fig. 1a) to a final state (e.g., Fig. 1b) by means of an operational semantics [13].



(a) Initial state of the system. (b) Final state of the system.

Fig. 1. A containerization example.

This paper presents the *real-time sccp* calculus (rtsccp), a generalization of sccp with timing constraints. The intended models of rtsccp are spatially-distributed multi-agent reactive systems that may have different computing

capabilities (e.g., virtual containers with heterogeneous bandwidth and main memory configurations) and be subject to real-time requirements (e.g., upper bounds in the execution time of a given operation). The formal semantics of rtsccp is given in the form of a real-time rewriting logic semantics that is executable in the Maude system [5]. As such, the real-time rewriting logic specification can be subject to automatic reachability analysis and LTL model-checking, and thus enables the formal analysis of timing behavior for agents distributed in hierarchical spaces, such as fault-tolerance and consistency.

In the rtsccp real-time rewriting logic semantics, flat configurations of object-like terms encode the hierarchical structure of spaces, and equational and rewrite rules both axiomatize the concurrent computational steps of processes. Time attributes are associated to process-store interaction, as well as to process mobility in the space structure, by means of maps from agents to non-negative real quantities; these choices can be interpreted to denote, as previously mentioned, upper bounds in the execution time of the given operations. The underlying constraint system of sccp is materialized with the help of the rewriting modulo SMT [21] approach, with constraints being quantifier-free formulas over Boolean and integer shared variables, and information entailment queried as semantic inference and automatically delivered by the SMT-based decision procedures. The main contribution of this work can also be seen as yet another interesting use of rewriting logic as a semantic framework: the support in rewriting logic for real-time systems [18] and open systems [21], make of the rtsccp real-time rewriting logic semantics a symbolic and fully executable specification in Maude, that is both sound and complete (relative to initial semantics) for reachability analysis in spatial constraint systems with discrete and dense linear timing constraints.

Outline. Section 2 summarizes some preliminaries on constraint systems and rewriting logic. The real-time rewriting logic semantics for rtsccp is presented in Sect. 3. The use of the semantics for symbolic reachability analysis is illustrated with some examples in Sect. 4. Additional related work and concluding remarks are included in Sect. 5. The rewriting logic specification for rtsccp and the examples are available at

http://escher.javerianacali.edu.co/rtsccpe/index.html

2 Preliminaries

This section briefly explains the basics of constraint and spatial constraint systems with extrusion. It also presents a summary of order-sorted rewriting logic [14], a semantic framework that unifies a wide range of models of concurrency.

A *constraint system* (cs) $\mathbf{C}$ is a complete algebraic lattice $\mathbf{C} = (Con, \sqsubseteq)$, where the elements in *Con* are called *constraints* and the order $\sqsubseteq$ *entailment of information*: $d \sqsubseteq c$ (or, $c \sqsupseteq d$) asserts that the constraint c contains at least as much information as the constraint d. The symbols $\sqcup$, *true*, and *false* denote

the least upper bound (*lub*) operation, the bottom, and the top element of $\mathbf{C}$, respectively. An n-agent *spatial constraint system* ($n-\mathsf{scs}$) [13] $\mathbf{C}$ is a cs $(Con, \sqsubseteq)$ equipped with n self-maps $[\cdot]_1, \ldots, [\cdot]_n$ over its set of constraints Con satisfying, for each function $[\cdot]_i : Con \to Con$: $[true]_i = true$ and $[c \sqcup d]_i = [c]_i \sqcup [d]_i$, for each $c, d \in Con$. An *n-agent spatial constraint system with extrusion (n-scse)* [12] is an n-scs $\mathbf{C}$ equipped with n self-maps $\uparrow_1, \ldots, \uparrow_n$ over Con, written $(\mathbf{C}, \uparrow_1, \ldots, \uparrow_n)$, such that each $\uparrow_i$ is the right inverse of $[\cdot]_i$.

A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E \uplus B, R)$ with: (i) $(\Sigma, E \uplus B)$ an order-sorted equational theory with signature Σ, E a set of equations over T_Σ, and B a set of structural axioms – disjoint from the set of equations E – over T_Σ for which there is a finitary matching algorithm (e.g., associativity, commutativity, and identity, or combinations of them); and (ii) R a finite set of (possibly conditional) rewrite rules over $T_\Sigma(X)$. Intuitively, $\mathcal{R}$ specifies a concurrent system whose states are elements of the set $T_{\Sigma/E\uplus B}$ of Σ-terms modulo $E \uplus B$ and whose concurrent transitions are axiomatized by the rules R according to the inference rules of rewriting logic [3]. In particular, for $t, u \in T_\Sigma$ representing states of the concurrent system described by $\mathcal{R}$, a transition from t to u is captured by a formula of the form $t \to_\mathcal{R} u$; the symbol $\to_\mathcal{R}$ denotes the binary rewrite relation induced by R over $T_{\Sigma/E\uplus B}$ and $\mathcal{T}_\mathcal{R} = (T_{\Sigma/E\uplus B}, \to_\mathcal{R})$ denotes the *initial reachability model* of $\mathcal{R}$.

The rewriting logic semantics of a language $\mathcal{L}$ is a rewrite theory $\mathcal{R}_\mathcal{L} = (\Sigma_\mathcal{L}, E_\mathcal{L} \uplus B_\mathcal{L}, R_\mathcal{L})$ where $\to_{\mathcal{R}_\mathcal{L}}$ provides a step-by-step formal description of $\mathcal{L}$'s *observable* run-to-completion mechanisms. The conceptual distinction between equations and rules in $\mathcal{R}_\mathcal{L}$ has important consequences that are captured by rewriting logic's *abstraction dial* [15]. Setting the level of abstraction in which all the interleaving behavior of evaluations in $\mathcal{L}$ is observable, corresponds to the special case in which the dial is turned down to its minimum position by having $E_\mathcal{L} \uplus B_\mathcal{L} = \emptyset$. The abstraction dial can also be turned up to its maximal position as the special case in which $R_\mathcal{L} = \emptyset$, thus obtaining an equational semantics of $\mathcal{L}$ without observable transitions. The rewriting logic semantics presented in this paper is *faithful* in the sense that such an abstraction dial is set at a position that exactly captures the interleaving behavior of the concurrency model.

The real-time rewrite theory presented in this work is *time-robust*, namely: (i) in any given state, time can advance either any amount up to a specific instant in time or not at all; and (ii) instantaneous rules (i.e., those that are not tick rules and are supposed to take zero time) can only be applied when the system has advanced the maximal possible amount of time before any timed action can become enabled. Under these two assumptions and by using the maximal time sampling strategy, unbounded and time-bounded search and model checking are sound and complete with respect to *timed fair paths* [19]. They exclude paths with an infinite sequence of tick steps where, at each step, time could have advanced to time r (the duration of the first step in a path) or beyond, but with a total path duration less than r. Also are excluded those 'unfair' paths containing an infinite and consecutive sequence of 0-time ticks over a state on which an instantaneous rule can be applied. Note that a time-robust system may have Zeno paths, where the sum of the durations of an infinite number of tick

steps is bounded. By restricting the computations to time-bounded prefixes only a finite set of states can be reached from an initial state, so that the target real-time specification does not exhibit any Zeno behavior and temporal properties can be model checked.

Satisfiability Modulo Theories (SMT) studies methods for checking satisfiability of first-order formulas in specific models. In this work, the representation of the constraint system is based on SMT solving technology. Given an many-sorted equational theory $\mathcal{E}_0 = (\Sigma_0, E_0)$ and a set of variables $X_0 \subseteq X$ over the sorts in Σ_0, the formulas under consideration are in the set $QF_{\Sigma_0}(X_0)$ of quantifier-free Σ_0-formulas: each formula being a Boolean combination of Σ_0-equation with variables in X_0 (i.e., atoms). The terms in $T_{\Sigma_0/E_0}(X)$ are called *built-ins* and represent the portion of the specification that will be handled by the SMT solver (i.e., they are semantic data types). In this setting, an SMT instance is a formula $\phi \in QF_{\Sigma_0}(X_0)$ and the initial algebra $\mathcal{T}_{\mathcal{E}_0^+}$, where $\mathcal{E}_0^+$ is a *decidable extension* of $\mathcal{E}_0$ such that ϕ is satisfiable in $\mathcal{T}_{\mathcal{E}_0^+}$ iff there exists $\sigma : X_0 \longrightarrow T_{\Sigma_0}$ such that $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$.

Maude [5] is a language and tool supporting the formal specification, execution, and analysis of concurrent systems specified as rewrite theories, including those with real-time semantics (see [19]) and those with built-ins as proposed in the rewriting modulo SMT approach (see [21]).

3 Rewriting Logic Semantics

This section introduces the rtsccp real-time rewriting logic semantics in the form of a real-time rewrite theory $\mathcal{R}$, detailing some aspects of to its syntax and transitions.

Figure 2 depicts the module structure of $\mathcal{R}$, where a triple-line arrow ($\Rrightarrow$) represents module importation by protecting and a single-line arrow ($\rightarrow$) module importation by inclusion. The difference between these two importing modes is that the former allows surjectivity (*junk*) and injectivity (*confusion*) [5].

3.1 System States

The tree-like structure of the hierarchical spaces is represented as a flat configuration of object-like terms encoding the state of execution of the agents. The hierarchical relationships among spaces are specified by common prefixes as part of an agent's name. In an observable state, each agent's space is represented by a set of object-like terms: some encoding the state of execution of all its processes and exactly one object representing its local store. The object-based system is represented using Maude's predefined module CONFIGURATION imported in SCCP-STATE. The object and class identifiers are:

```
subsorts Nat Aid < Oid .
ops agent process Simulation :    -> Cid .
op {_}         : Configuration    -> Sys .
```

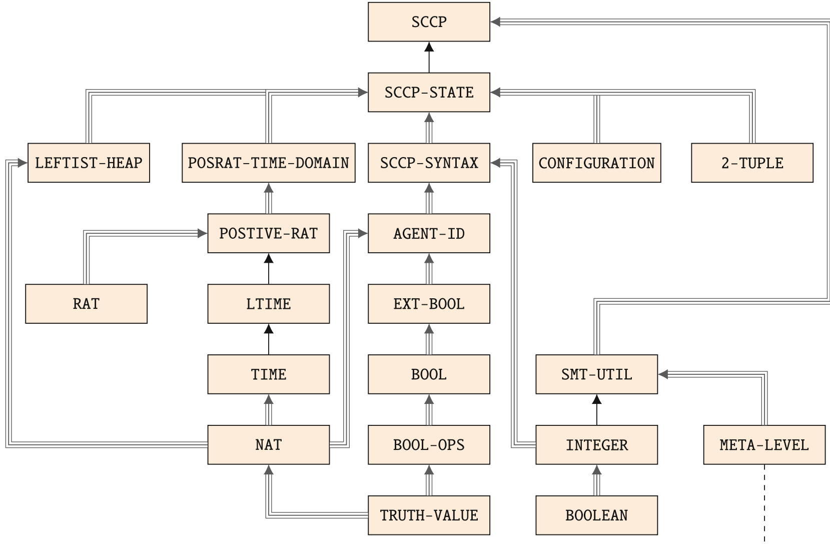


Fig. 2. Module hierarchy of rtsccp

The system states are represented by the topsort `Sys` with argument a configuration of objects containing the setup of each one of the agents in the system. A `Configuration` is a multiset of objects with set union denoted by juxtaposition and identity `none`. There are two types of object identifiers: agent identifiers (`Aid`) for identifying agents and their hierarchical structure, and natural numbers (`Nat`) for some additional identification used internally in $\mathcal{R}$. There are three types of class identifiers, namely, for agents, processes, and a simulation object. A simulation object specifies the attributes required for the real-time simulation of the system, such as the global time and the scheduler.

Each agent has one attribute, namely, its a store, and each process has two attributes: an universal identifier (used internally for execution purposes) and the command (i.e., process) it is executing:

```
op store :_    : Boolean        -> Attribute [ctor] .
op Uid :_      : Nat            -> Attribute [ctor] .
op command :_  : SCCPCmd        -> Attribute [ctor] .
```

The syntax of commands is presented in Sect. 3.2. Section 3.3 explains how quantifier free formulas are used to represent constraints (as `Boolean`) and the entailment relation is encoded with the help of SMT-based decisions procedures.

Finally, the attributes of the simulation object include the global time (attribute `gtime`); the priority queue of the system commands to be processed, ordered by time-to-execution (attribute `pqueue`); the collection of pending commands, i.e., ask commands that are waiting for its guarding constraint to become activate (attribute `pend`); the counter for assigning the next internal identifier when spawning a new process (attribute `nextID`); a flag that is on whenever a

tick rule needs to be applied (attribute `flg`); and a collection of maps containing the time it takes to process certain commands relative to the space where they are executed (attributes `MAsk`, `MTell`, `MSp`, and `MExt`). The sort `Time`, as it is often the case in Real-time Maude [18], can be used to represent either discrete or dense linear time, while `Ttime` is the name of the Maude view that is used to instantiate parameterized sorts with time:

```
op gtime :_     : Time              -> Attribute [ctor] .
op pqueue :_    : Heap{Tuple}       -> Attribute [ctor] .
op pend :_      : Heap{Tuple}       -> Attribute [ctor] .
op nextID :_    : Nat               -> Attribute [ctor] .
op flg :_       : Bool              -> Attribute [ctor] .
op MAsk :_      : Map{Aid, Ttime}  -> Attribute [ctor] .
op MTell :_     : Map{Aid, Ttime}  -> Attribute [ctor] .
op MSP :_       : Map{Aid, Ttime}  -> Attribute [ctor] .
op MExt :_      : Map{Aid, Ttime}  -> Attribute [ctor] .
```

As mentioned before, the qualified identifiers of agents are used to encode the hierarchical structure of spaces (sort `Aid`). The root of any tree is denoted by constant `root` and any other qualified name corresponds to a dot-separated list natural numbers (sort `Nat`), organized from left to right:

```
op root : -> Aid .
op _._  : Nat Aid -> Aid .
```

Example 1. In this syntax, the container system depicted in Fig. 1a can be specified as follows:

```
< root : agent | store : (W:Integer === (9).Integer) >
< 0 . root : agent | store : (X:Integer >= 11) >
< 0 . 1 . root : agent | store : (Y:Integer > 5) >
< 1 . root : agent | store : true >
< 2 . root : agent | store : true >
```

3.2 Commands

The following EBNF-like notation defines the process-like syntax of commands:

$$P ::= \mathbf{0} \mid \mathbf{tell}(c) \mid \mathbf{ask}(c) \rightarrow P \mid P \parallel P \mid [P]_i \mid \uparrow_i (P)$$

where c is a constraint and i an agent identifier. The $tell(c)$ command posts the constraint c to the local store (once a constraint is added, it cannot be removed from the store so that the store grows monotonically). The command $ask(c) \rightarrow P$ queries if c can be deduced from the information in the local store; if so, the agent behaves like P, otherwise, it remains blocked until more information is added to the store. A basic `ccp` process like-language usually adds *parallel composition* ($P \parallel Q$) combining processes P and Q concurrently. The command $[P]_i$ indicates that command P must be executed inside the agent i's space: any information

that P produces is available to other commands that execute within the same space. The command $\uparrow_i (P)$ denotes that P is to be run outside the space of agent i and the information posted by P is going to be stored in the parent of agent i. The SCCP-SYNTAX module includes the syntax of commands in rtsccp:

```
op 0       :                   -> SCCPCmd .
op tell    : Boolean           -> SCCPCmd .
op ask_->_ : Boolean SCCPCmd -> SCCPCmd .
op _||_    : SCCPCmd SCCPCmd -> SCCPCmd [assoc comm gather (e E) ] .
op _in_    : SCCPCmd Nat       -> SCCPCmd .
op _out_   : SCCPCmd Nat       -> SCCPCmd .
```

3.3 Time Scaffolding

The real-time behavior in $\mathcal{R}$ associates timing behavior to those commands that interact with stores (i.e., tell and ask commands) and to commands that involve mobility among the space structure of the system (i.e., $[_]_{_}$ and $\uparrow_{_}(_)$). More precisely, tell and ask commands can take time when posting and querying, respectively, from a store. Moving the execution of a command inside an agent and extruding from a space can also take up time. Such times are given by the time maps MTell (for *tell*), MAsk (for *ask*), MSp (for $[_]_{_}$), and MExt (for $\uparrow_{_}(_)$), and can be consulted using the getTimeCmd function. For example, MTell(i) denotes the time it takes to execute a tell command inside the agent's i space.

```
op fTime : Map{Aid, Ttime} Aid -> Time .
eq fTime(M:Map{Aid, Ttime}, L1)
 = if $hasMapping(M, L1) then M[L1] else 0 fi .

op getTimeCmd : Attribute Attribute Attribute Attribute
                                                SCCPCmd Aid -> Time .
eq getTimeCmd(MTell: MT, MAsk: MA, MIn: MI, MOut: MO, tell(B1), L1)
 = fTime(MT, L1) .
eq getTimeCmd(MTell: MT, MAsk: MA, MIn: MI, MOut: MO, C1 in I1, L1)
 = fTime(MI, L1) .
eq getTimeCmd(MTell: MT, MAsk: MA, MIn: MI, MOut: MO, C1 out I1, L1)
 = fTime(MO, L1) .
eq getTimeCmd(MTell: MT, MAsk: MA, MIn: MI, MOut: MO, C1, L1)
 = 0 [owise] .
```

The run-to-completion time of commands is simulated with the help of a leftist heap that keeps track of all the active commands that are waiting for the global timer to advance. One motivation to use leftist heaps is that insertion, removal, and querying are defined without the need of structural axioms, which may result in performance gains during execution. A *leftist heap* [17] is a heap-ordered binary tree that satisfies the *leftist property*: the rank (i.e., the length of its rightmost path to a leaf) of any left child is at least as large as the rank of its right sibling. Each entry in the heap is a pair (i, t) where i is an unique identifier of a process and t the time it needs to start executing. At the beginning,

all the processes belong to the heap and they are ordered with respect to their execution time. A process is executed when its execution time is the minimum time of all the processes that are pending to complete their transitions. The leftist heap is implemented as a parameterized container in the functional module LEFTIST-HEAP{X :: STRICT-TOTAL-ORDER}, with admissible parameters only being strict total orders:

```
sort Heap{X} NeHeap{X} .
subsort NeHeap{X} < Heap{X} .
op empty : -> Heap{X} [ctor] .
op T(_,_,_,_) : Nat X$Elt Heap{X} Heap{X} -> NeHeap{X} [ctor] .
op isEmpty : Heap{X} -> Bool .
eq isEmpty(empty)       = true .
eq isEmpty(T(Ra,E,L,R)) = false .
op rank    : Heap{X} -> Nat .
eq rank(empty)       = 0 .
eq rank(T(Ra,E,L,R)) = Ra .
op makeT   : X$Elt Heap{X} Heap{X} -> NeHeap{X} .
eq makeT(E,L,R)
 = if rank(L) >= rank(R)
   then T(rank(R) + 1,E,L,R)
   else T(rank(L) + 1,E,R,L)
   fi .
```

Heaps are constructed from the constant empty and the T(_,_,_,) function symbol: the first argument is the rank of the tree (sort Nat), the second one the label (sort X$Elt), and the third and fourth ones the left and right children (sort Heap{X}), respectively. In the semantics of rtsccp, the sort X$Elt is instantiated with the sort of pairs of the form (I, T), where I is an internal process identifier and T is the run-completion time of such a process. Auxiliary operations include isEmpty, rank, and makeT, which are used to verify whether a heap is empty, compute the rank of a given heap, and create a heap out of two heaps, respectively. Other key operations on leftist heaps are the merging of two heaps (function merge), inserting an element in a heap (function insert), removing an element from a heap (function deleteMin), and finding the element at the top of a non-empty heap (function findMin).

```
op merge     : Heap{X} Heap{X} -> Heap{X} .
eq merge(empty, L) = L .
eq merge(L, empty) = L .
eq merge(T(Ra,E,L,R),T(Ra',E',L',R'))
 = if (E < E' or E == E')
   then makeT(E,L,merge(R,T(Ra',E',L',R')))
   else makeT(E',L',merge(T(Ra,E,L,R),R'))
   fi .
op insert    : X$Elt Heap{X} -> NeHeap{X} .
eq insert(E,L) = merge(T(1,E,empty,empty),L) .
op deleteMin : NeHeap{X} -> Heap{X} .
eq deleteMin(T(Ra,E,L,R)) = merge(L,R) .
```

```
op findMin   : NeHeap{X} -> X$Elt .
eq findMin(T(Ra,E,L,R)) = E .
```

3.4 The Constraint System

In this rewriting logic semantics, the sort `Boolean` (available in the current version of Maude from the `INTEGER` module) defines the data type used to represent rtsccp's constraints. Terms of sort `Boolean` are quantifier-free formulas built from variables ranging over the Booleans and integers, and the usual function symbols. The current version of Maude is integrated with the CVC4 [2] and Yices2 [8] SMT solvers, which can be queried via the meta-level. In this semantics, queries to the SMT solvers are encapsulated by functions `check-sat` and `check-unsat`:

```
op check-sat : Boolean -> Bool .
eq check-sat(B) = metaCheck(['INTEGER], upTerm(B)) .
op check-unsat : Boolean -> Bool .
eq check-unsat(B) = not(check-sat(B)) .
```

The function invocation `check-sat`(B) returns true only if B is satisfiable. Otherwise, it returns false if it is unsatisfiable or undefined if the SMT solver cannot decide. Note that function invocation `check-unsat`(B) returns true only if B is unsatisfiable. Therefore, the rewriting logic semantics of rtsccp instantiates the constraint system $\mathbf{C} = (Con, \sqsubseteq)$ by having quantifier-free formulas, modulo the semantic equivalence in $\mathcal{T}_{\mathcal{E}_0^+}$ (i.e., the model implemented in the SMT solver extending the initial model $\mathcal{T}_{\mathcal{E}_0}$), as the constraints Con and semantic validity relative to $\mathcal{T}_{\mathcal{E}_0^+}$ as the entailment relation $\sqsubseteq$. More precisely, if Γ is a finite set of terms of sort `Boolean` and ϕ is term of sort `Boolean`, the following equivalence holds: $\Gamma \sqsubseteq \phi$ iff `check-unsat` $\left(\left(\bigwedge_{\gamma \in \Gamma} \gamma\right) \wedge \neg\phi\right)$. In order to make a direct relation between the entailment relation $\sqsubseteq$ and the Maude syntax, the operator `entails` is defined as follows:

```
op entails : Boolean Boolean -> Bool .
eq entails(C1:Boolean, C2:Boolean)
 = check-unsat(C1:Boolean and not(C2:Boolean)) .
```

3.5 System Transitions

The tick rule models time elapse in the system [18]:

```
crl [tick] :
  { X < I : Simulation | pqueue : P, gtime : T, flg : true,
            pend : P0, Atts > }
 => { X < I : Simulation | pqueue : merge(delta(deleteMin(P),T0),P0),
              gtime : (T plus T0), flg : false, pend : empty, Atts > }
 if T0 := p2(findMin(P)) .
```

where the the auxiliary operation `delta` reduce `T0` units the execution time of every command in the heap `P`:

```
op delta : Heap{Tuple} Time -> Heap{Tuple} .
eq delta(empty,T') = empty .
eq delta(T(N,((I,T)),P,P0),T')
 = T(N,((I,T monus T')),delta(P,T'),delta(P0,T')) .
```

When the `[tick]` rule is fired, the global time is incremented in `T0` units, where `T0` is the minimum time present in the priority queue `P`, which is modified by removing the process with the minimum execution time. It also adds the pending commands to the priority queue. The pending commands are ask commands that, although they have been activated already for execution, have not been able to execute because their guard has not been met by the state of the corresponding local stores. The tick rule puts all these pending process back in the main queue, so that their guards can be checked again and be executed or put back in the pending queue. Figure 3 depicts the possible transitions an ask command can take between being in the priority queue, in the pending queue, and finally executing. The rules `[ask]` and `[delay]` are introduced below.

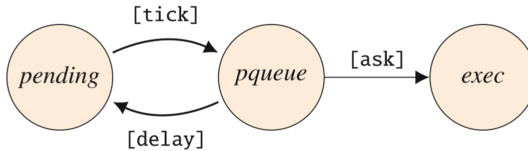


Fig. 3. Possible transitions for ask commands.

The invisible transitions of the semantics are specified with the help of equational rules. Namely, one for removing a `0` command from a configuration and another one to join the contents of two stores of the same space (i.e., two stores with the same `Aid`). The latter type of transition is important because when a new process is spawned in a agent's space, a store with the empty constraint (i.e., `true`) is created for that space. If such a space existed before, then the idea is that the newly created store is subsumed by the existing one. Note that neither of the invisible transitions takes time, i.e., they are really instantaneous, and they axiomatize structural properties of commands.

```
eq { < L0: process | command: 0, Atts > X }
 = { none X } .
eq < L0: agent | store: B0 > < L0: agent | store: B1 >
 = < L0: agent | store: (B0 and B1) > .
```

The following six rules capture the concurrent observable behavior in $\mathcal{R}$:

```
crl [tell]:
    { < L0: agent | store: B0 >
```

```
      < L0: process | UID: I0, command: tell (B1) >
      < I: Simulation | pqueue: H, flg: false, pend: P, Atts > X }
 => { < L0: agent | store: (B0 and B1) >
      < I: Simulation | pqueue: H, flg: true, pend: P, Atts > X }
 if I0 == p1(findMin(H)) .

crl [parallel]:
    { < L0: process | UID: I0, command: (C0 || C1) >
      < I: Simulation | pqueue: H, nextID: N, flg: false, pend: P,
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 => { < L0: process | UID: N, command: C0 >
      < L0: process | UID: (N + 1), command: C1 >
      < I: Simulation | pqueue: H, nextID: (N + 2), flg: true,
        pend: H0, MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 if I0 == p1(findMin(H))
 /\ H0 := insert(((N, getTimeCmd(MTell: MT, MAsk: MA, MIn: MI,
                             MOut: MO, C0, L0))),
         insert(((N + 1, getTimeCmd(MTell: MT, MAsk: MA, MIn: MI,
                                 MOut: MO, C1, L0))), P)) .

crl [space]:
    { < L0: process | UID: I0, command: (C0 in N0) >
      < I: Simulation | pqueue: H, nextID: N, flg: false, pend: P,
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 => { < N0 . L0: agent | store: true >
      < N0 . L0: process | UID: N, command: C0 >
      < I: Simulation | pqueue: H, flg: true, pend: H0, nextID: (N+1),
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 if I0 == p1(findMin(H))
 /\ H0:= insert(((N, getTimeCmd(MTell: MT,
               MAsk: MA, MIn: MI, MOut: MO, C0, L0))), P) .

crl [extrussion]:
    { < N0 . L0: process | UID: I0, command: (C0 out N0) >
      < I: Simulation | pqueue: H, nextID: N, flg: false, pend: P,
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 => { < L0: process | UID: N, command: C0 >
      < I: Simulation | pqueue: H, flg: true, pend: H0, nextID: (N+1),
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 if I0 == p1(findMin(H))
 /\ H0:= insert(((N, getTimeCmd(MTell: MT,
                 MAsk: MA, MIn: MI, MOut: MO, C0, N0 . L0))), P) .

crl [ask]:
    { < L0: agent | store: B0 >
      < L0: process | UID: I0, command: (ask B1 -> C1) >
      < I: Simulation | pqueue: H, flg: false, pend: P, nextID: N,
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 => { < L0: agent | store: B0 >
      < L0: process | UID: N, command: C1 >
```

```
      < I: Simulation | pqueue: H, flg: true, pend: H0, nextID: (N+1),
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
  if I0 == p1(findMin(H))
  /\ entails(B0,B1)
  /\ H0:= insert(((N,getTimeCmd(MTell: MT, MAsk: MA, MIn: MI,
                     MOut: MO, C1, L0) plus alpha(MA, L0))), P) .

 crl [delay]:
    { < L0: agent | store: B0 >
      < L0: process | UID: I0, command: (ask B1 -> C1) >
      < I: Simulation | pqueue: H, pend: P, Atts > X }
  => { < L0: agent | store: B0 >
      < L0: process | UID: I0, command: (ask B1 -> C1) >
      < I: Simulation | pqueue: deleteMin(H),
                          pend: insert(((I0,T1)),P),Atts > X }
  if I0 == p1(findMin(H))
  /\ not(entails(B0,B1))
  /\ T1:= (p2(findMin(H))) .
```

The [tell] rule implements the execution semantics of a tell command by posting the given constraint in the local store and by removing such a command from the configuration. The [parallel] rule implements the semantics for parallel composition of process by spawning the two process in the current space. Rule [space] creates a new agent's space denoted by N0.L0, with an empty store (i.e., true), beside the execution of program C0 within the new agent's space. The [extrusion] rule executes C0 in the parent's space L0. The [ask] rule executes a command C1 when the guard B1 in ask B1 -> C1 holds: that is, when B1 is a semantic consequence of the contents B0 of the local store. Note that the semantic consequence relation of the constraint system is queried by asking the SMT solver. The [delay] rule represents the negative answer for the ask rule: whenever B0 does not entail B1 or B1 is not a semantic consequence of the contents B0 of the store. The [delay] rule moves an ask command into the pending heap, where it will remain until the tick rule executes again.

4 Reachability Analysis

The goal of this section is to explain how the rewriting logic semantics $\mathcal{R}$ of rtsccp and rewriting modulo SMT can be used as an automatic mechanism for solving existential reachability goals in the initial model $\mathcal{T}_\mathcal{R}$. This approach can be useful for symbolically proving or disproving real-time safety properties of $\mathcal{T}_\mathcal{R}$. The approach presented in this section mainly relies on Maude's search command, but it can be easily extended to be useful in the more general setting of Maude's LTL Model Checker.

The examples presented in this section use the following time functions for the processes:

```
MAsk  : root |-> 1/20, (0 . root) |-> 1/10, (0 . 1 . root) |-> 3/20,
        (1 . root) |-> 1/10, (2 . root) |-> 1/10
```

```
MTell : root |-> 1/10, (0 . root) |-> 3/20, (0 . 1 . root) |-> 1/5,
        (1 . root) |-> 3/20, (2 . root) |-> 3/20
MSP   : root |-> 1/2, (0 . root) |-> 7/10, (0 . 1 . root) |-> 4/5,
        (1 . root) |-> 13/20, (2 . root) |-> 3/5
MExt  : root |-> 1/2, (0 . root) |-> 13/20, (0 . 1 . root) |-> 1,
        (1 . root) |-> 1/2, (2 . root) |-> 3/5
```

For example, querying the store at the root agent takes 1/20 time units.

Fault tolerance is the property that ensures a system to continue operating properly in the event of the failure; *consistency* means that a local failure does not propagate to the entire system. In $\mathcal{R}$, this means that if a store becomes inconsistent, it is not the case that such an inconsistency spreads to the entire system. Of course, inconsistencies can appear in other stores due to some unrelated reasons.

Searching an inconsistent store can be easily implemented with the help of $\mathcal{R}$ and Maude's `search` command. The answer of this command in the positive would mean that from some initial state, there is a state in which a store becomes inconsistent at some point of execution within a given time interval. Taking advantage of $\mathcal{R}$ and the rewriting modulo SMT approach, also is possible to know when a store is inconsistent. As an example, consider the container system in Fig. 1a and the following `search` command with a the time interval $[0, 1.5)$:

```
search in SCCP :
    { < root : agent | store : (W:Integer === (9).Integer) >
      < 0 . root : agent | store : (X:Integer >= 11) >
      < 0 . 1 . root : agent | store : (Y:Integer > 5) >
      < 1 . root: agent | store: true > < 2 . root: agent | store: true >
      < root : process | UID : 1, command : (((ask X:Integer > 2 ->
        (tell(Y:Integer < 10) in 0 in 1 out 0)) in 0)
        || ((tell(Z:Integer =/== (10).Integer)
        || (tell(T:Integer === 1) in 3)) in 2))
        || (tell(X:Integer <= 10) in 0)) >
      < 1 : Simulation | gtime : 0,pqueue : T(1,((1,0)),empty,empty),
        pend : empty,nextID : 19, flg : false, ... (time maps)}
=>* { < 1 : Simulation | gtime : T:Time, Atts:AttributeSet >
      < A:Aid : agent | store : B0:Boolean > C:Configuration }
    such that check-unsat(B0:Boolean) and T < 3/2 .
```

Note that a store is inconsistent if it is unsatisfiable, thereby checking whether a store is inconsistent is accomplished with the function `check-unsat`. The aforementioned command searches for an inconsistent store during the first 1.5 units of time of the system's execution. This command does not find an inconsistent store between the first 1.5 units of time in any of the 56 reachable states. However, it is possible to make a store inconsistent by adding inconsistent information, for example by adding the process `tell(X <= 10) in 0`. The output for the `search` command is:

```
Solution 1 (state 159)
states: 160  rewrites: 16666 in 876ms cpu (875ms real)
```

```
        (19025 rewrites/second)
C:Configuration --> < root:agent | store:(W:Integer === (9).Integer) >
  < 0 . 1 . root : agent | store : (Y:Integer > 5) >
  < 1 . root:agent | store:true > < 2 . root:agent | store:true >
  < 2 . root:process | UID: 28,
                       command: tell(Z:Integer =/== (10).Integer) >
  < 2 . root:process | UID: 27,
                       command: (tell(T:Integer === (1).Integer) in 3) >
  < 0 . root:process | UID : 24,
                       command: (tell(Y:Integer < 10) in 0 in 1 out 0) >
A:Aid --> 0 . root
B0 --> X:Integer >= (11).Integer and X:Integer <= (10).Integer
Atts --> pqueue : T(2,(25,1/20),T(1,(24,4/5),empty,empty),T(1,(28,1/10),
  T(1,(27,3/5),empty,empty),empty)),pend : empty,nextID : 29,
  flg : true, ... (time maps)
T --> 1/2
```

There are 238 reachable states (from the initial state) and 74 of them have an inconsistent store between the first 1.5 units of time. The first inconsistency appears in 0.5 time units, and the last one in 1.3 times units. Note that, the system continues evolving even though there is an inconsistency. It is possible to verify that there are states with consistent and inconsistent stores at the same time by slightly modifying the above search command.

Knowledge inference refers to acquiring new knowledge from existing facts. In the setting of $\mathcal{R}$, this means that from a given initial state an agent, at some point, has gained enough information to infer new facts. A positive answer to such a query, means that from some initial state, at some moment during execution, there is at least one agent that has gained enough information to infer the given facts. As an example, consider the container system in Fig. 1a and the following `search` command:

```
search in SCCP :
    { < root : agent | store : (W:Integer === (9).Integer) >
      < 0 . root : agent | store : (X:Integer >= 11) >
      < 0 . 1 . root : agent | store : (Y:Integer > 5) >
      < 1 . root : agent | store : true >
      < 2 . root : agent | store : true >
      < root : process | UID : 1, command : (((ask X:Integer > 2 ->
        (tell(Y:Integer < 10) in 0 in 1 out 0)) in 0)
        || ((tell(Z:Integer =/== (10).Integer)
        || (tell(T:Integer === 1) in 3)) in 2)) >
      < 1 : Simulation | gtime : 0,pqueue : T(1,((1,0)),empty,empty),
        pend : empty, nextID : 19, flg : false, ... (time maps) > }
=>* { < 1 : Simulation | gtime : T:Time, Atts:AttributeSet >
      < A:Aid : agent | store : B0:Boolean > C:Configuration }
    such that entails(B0:Boolean, Y:Integer < 15) and T:Time > 0
              and T:Time < 2 .
```

It checks if there is a state, reachable from the given initial state, in which some store logically implies $Y < 15$ in the time interval $(0, 2)$. This command does not

find a container with enough information in such time interval in any of the 56 reachable states. However, if the time interval in the command is changed to $(2, 3)$ the query finds two solutions:

```
Solution 1 (state 54)
states: 55  rewrites: 7466 in 360ms cpu (358ms real)
                      (20738 rewrites/second)
C:Configuration --> < root:agent | store:(W:Integer === (9).Integer) >
  < 0 . root : agent | store : (X:Integer >= 11) >
  < 1 . root : agent | store : true >
  < 2 . root : agent | store : (Z:Integer =/== (10).Integer) >
  < 3 . 2 . root : agent | store : (T:Integer === (1).Integer) >
A:Aid --> 0 . 1 . root
B0:Boolean --> Y:Integer > (5).Integer and Y:Integer < (10).Integer
Atts:AttributeSet --> pqueue : T(1,(29,1/10),empty,empty),pend : empty,
  nextID : 30,flg : true, ... (time maps)
T:Time --> 5/2

Solution 2 (state 55)
states: 56  rewrites: 7601 in 368ms cpu (366ms real)
                      (20654 rewrites/second)
C:Configuration --> < root:agent | store:(W:Integer === (9).Integer) >
  < 0 . root : agent | store : (X:Integer >= 11) >
  < 1 . root : agent | store : true >
  < 2 . root : agent | store : (Z:Integer =/== (10).Integer) >
  < 3 . 2 . root : agent | store : (T:Integer === (1).Integer) >
A:Aid --> 0 . 1 . root
B0:Boolean --> Y:Integer > (5).Integer and Y:Integer < (10).Integer
Atts:AttributeSet --> pqueue : empty,pend : empty,nextID : 30,
  flg : false, ... (time maps)
T:Time --> 13/5
...
```

The webpage at

http://escher.javerianacali.edu.co/rtsccpe/index.html

contains more details about this example and other examples about reachability analysis with $\mathcal{R}$, including knowledge inference and equivalence of knowledge.

5 Related Work and Concluding Remarks

In addition to the related work included in Sect. 1, it is important to mention other related research in the area of timing semantics for concurrent constraint programming. An extension of concurrent constraint programming in [24] presents a timed asynchronous computation model and propose an implementation using loop-free deterministic finite automata, a declarative framework for reactive systems where time is represented as discrete time units. More recently, Pérez and Rueda [20] propose an operational semantics based on probabilistic

automaton, extending the work in [24], with probabilistic and non-deterministic choices for processes. The inclusion of stochastic information for processes proposed by Aranda et al. in [1] associates to each computation a random variable determining its time duration: given a set of competing actions, the fastest action is executed, that is, the one with the shortest duration. Finally, Sarria and Rueda [27] present a real-time extension of ccp with application to music interaction.

In the realm of rewriting logic, Degano et al. [7] provide a rewriting logic semantics for Milner's CCS with interleaving behavior. Additionally, a set of axioms is defined for a logical characterization of the concurrency of CCS processes. In [28], the authors use rewriting logic to represent the semantics of CCS and a modal logic for describing local capabilities of CCS processes. In particular, they study how to make executable the SOS semantics of CSS and present a fully executable specification of the semantics. More recently, Romero and Rocha [22] have proposed a symbolic rewriting logic semantics of the spatial modality of ccp with extrusion.

This paper has presented a real-time rewriting logic semantics for spatial concurrent constraint programming (rtsccp) that is fully executable in the Maude system. The intended models of rtsccp are spatially-distributed multi-agent reactive systems that may have different computing capabilities and be subject to real-time requirements. In this setting, time attributes are associated to process-store interaction, as well as to process mobility in the space structure, by means of maps from agents to non-negative real quantities. Details about the underlying constraint system have been given as materialized with the help of rewriting modulo SMT. Furthermore, examples of reachability analysis performed on this semantics have been given to illustrate certain aspects of the timing behavior of agents distributed across hierarchical spaces, such as fault-tolerance and consistency.

Future work can span in two directions. One interesting direction to follow is to pursue challenging case studies in which, with the help of the real-time rewriting logic semantics for rtsccp presented in this work, other key aspects of spatially distributed concurrent processes such as privacy can be analyzed. The other direction, is to pursue a more general and fully symbolic rewriting logic semantics for rtsccp where time information can also be modeled as shared variables under the control of the SMT decision procedures. In such a setting, interesting properties of real-time systems such as missed deadlines and deadlocks could be fully analyzed, e.g., for infinitely many initial states in a system.

Acknowledgments. The authors would like to thank the anonymous referees for their helpful comments. The first author was partially supported by Colciencias via the project CLASSIC (Proj. No. 125171250031). The second author was partially supported by Colciencias' Convocatoria 761 Jóvenes Investigadores e Innovadores 2016 and Pontificia Universidad Javeriana Cali (Contract No. 416-2017). The third author was partially supported by Capital Semilla 2017, project "SCORES: Stochastic Concurrency in Rewrite-based Probabilistic Models" (Proj. No. 020100610). The third and fourth authors were partially supported by CAPES, Colciencias, and INRIA

via the STIC AmSud project "EPIC: EPistemic Interactive Concurrency" (Proj. No. 88881.117603/2016-01).

References

1. Aranda, J., Pérez, J.A., Rueda, C., Valencia, F.D.: Stochastic behavior and explicit discrete time in concurrent constraint programming. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 682–686. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89982-2_57
2. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
3. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theor. Comput. Sci. **360**(1–3), 386–414 (2006)
4. Chiarugi, D., Falaschi, M., Hermith, D., Marangoni, R., Olarte, C.: Stochastic modelling of non markovian dynamics in biochemical reactions. In: Rojas, I., Guzman, F.M.O. (eds.) International Work-Conference on Bioinformatics and Biomedical Engineering, IWBBIO 2013, Granada, Spain, 18–20 March 2013. Proceedings, pp. 537–544. Copicentro Editorial (2013)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
6. de Boer, F., Gabbrielli, M., Meo, M.C.: A timed concurrent constraint language. Inf. Comput. **161**, 45–83 (2000)
7. Degano, P., Gadducci, F., Priami, C.: A causal semantics for CCS via rewriting logic. Theor. Comput. Sci. **275**(1–2), 259–282 (2002)
8. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
9. Gilbert, D., Palamidessi, C.: Concurrent constraint programming with process mobility. In: Lloyd, J., et al. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 463–477. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44957-4_31
10. Gupta, V., Jagadeesan, R., Panangaden, P.: Stochastic processes as concurrent constraint programs. In: Symposium on Principles of Programming Languages, pp. 189–202 (1999)
11. Gupta, V., Jagadeesan, R., Saraswat, V.A.: Computing with continuous change. Sci. Comput. Program. **30**(1–2), 3–49 (1998)
12. Guzmán, M., Haar, S., Perchy, S., Rueda, C., Valencia, F.D.: Belief, knowledge, lies and other utterances in an algebra for space and extrusion. J. Log. Algebr. Methods Program. **86**(1), 107–133 (2017)
13. Knight, S., Palamidessi, C., Panangaden, P., Valencia, F.D.: Spatial and epistemic modalities in constraint-based process calculi. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 317–332. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_23
14. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992)
15. Meseguer, J., Roşu, G.: The rewriting logic semantics project: a progress report. Inf. Comput. **231**, 38–69 (2013)

16. Nielsen, M., Palamidessi, C., Valencia, F.D.: Temporal concurrent constraint programming: denotation, logic and applications. Nordic J. Comput. **9**(1), 145–188 (2002)
17. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1. paperback ed., transf. to digital printing edition) (2003). OCLC: 552279078
18. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theor. Comput. Sci. **285**(2), 359–405 (2002)
19. Ölveczky, P.C., Meseguer, J.: Abstraction and completeness for Real-Time Maude. Electron. Notes Theor. Comput. Sci. **176**(4), 5–27 (2007)
20. Pérez, J.A., Rueda, C.: Non-determinism and probabilities in timed concurrent constraint programming. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 677–681. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89982-2_56
21. Rocha, C., Meseguer, J., Muñoz, C.: Rewriting modulo SMT and open system analysis. J. Log. Algebr Methods Program. **86**(1), 269–297 (2017)
22. Romero, M., Rocha, C.: Symbolic execution and reachability analysis using rewriting modulo SMT for spatial concurrent constraint systems with extrusion. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NFM 2018. LNCS, vol. 10811, pp. 435–451. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_29
23. Saraswat, V., Jagadeesan, R., Gupta, V.: Foundations of timed concurrent constraint programming. In: Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science, pp. 71–80, 4–7 July 1994
24. Saraswat, V., Jagadeesan, R., Gupta, V.: Foundations of timed concurrent constraint programming, pp. 71–80. In: IEEE Computer Society Press (1994)
25. Saraswat, V., Jagadeesan, R., Gupta, V.: Default timed concurrent constraint programming. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 272–285, Jan 1995
26. Saraswat, V.A., Rinard, M., Panangaden, P.: Semantic foundations of concurrent constraint programming. In: POPL 1991, pp. 333–352. ACM (1991)
27. Sarria, G., Rueda, C.: Real-time concurrent constraint programming. In: 34th Latin American Conference on Informatics (CLEI 2008), pp. 379–391. CLEI (2008)
28. Verdejo, A., Martí-Oliet, N.: Two case studies of semantics execution in Maude: CCS and LOTOS. Form. Methods Syst. Design **27**(1–2), 113–172 (2005)

Approximating Any Logic Program by a CS-Program

Yohan Boichut(✉), Vivien Pelletier, and Pierre Réty

LIFO - Université d'Orléans, B.P. 6759, 45067 Orléans Cedex 2, France
{yohan.boichut,vivien.pelletier,pierre.rety}@univ-orleans.fr

Abstract. In this paper, we propose an extension of a technique transforming logic programs into a particular class of logic programs called CS-programs. Up to now, this technique is a semi-algorithm preserving the least Herbrand model. We integrate in this technique a process of generalization. Thanks to it, we are able to make the computation (the transformation) terminate and if we force the computation to terminate then we obtain a CS-program whose least Herbrand model contains the initial one. In this way, we can tackle successfully reachability problems that are out of the scope of techniques using regular approximations and also of the initial transformation technique (for which computations do not terminate).

Keywords: Logic program transformation · Reachability problem
Approximation · CS-program

1 Introduction

Some authors have introduced synchronized tree languages, initially expressed by grammar-like formalisms [4,6], and more recently expressed by some logic programs called CS-programs [7]. This class of languages is closed under intersection with a regular language and moreover, emptiness is decidable. A CS-program is a set of Horn clauses satisfying some constraints (bodies should be linear and without function symbols). In [8], the authors have proposed a semi-algorithm transforming any set of Horn clauses into a CS-program (recalled in Sect. 2). When the algorithm ends, one obtains a CS-program that recognizes exactly the same language, i.e. the same least Herbrand model, as the initial set of Horn clauses. However, the algorithm may not terminate, even if the initial set of Horn clauses could have been specified by a CS-program without loss of precision i.e. with the same Herbrand model.

In this paper, we propose an improvement of this technique, called *generalization*. Using generalization, we are able to make the initial technique always terminate. However the price to pay is that the obtained CS-program may have a language larger than the initial one (Sect. 3). Two examples are presented in Sect. 4. The former deals with a reachability problem that cannot be achieved

© Springer Nature Switzerland AG 2018
V. Rusu (Ed.): WRLA 2018, LNCS 11152, pp. 245–260, 2018.
https://doi.org/10.1007/978-3-319-99840-4_14

using a regular approximation [1]. The original semi-algorithm does not terminate whereas the language to construct is empty. We show that by using the generalization process, the procedure terminates and computes the empty language. The latter shows that we can perform program verification using the technique detailed in this paper. Moreover, the detailed example is out of the scope of techniques computing regular approximations (using tree automata in [3] or tree transducers in [2]). Before concluding, we discuss in Sect. 5 about the possibilities for improving this technique.

2 Preliminaries

Consider two disjoint sets, Σ a *finite ranked alphabet* and Var a set of variables. Each symbol $f \in \Sigma$ has a unique arity, denoted by $ar(f)$. The notions of *first-order term*, *position* and *substitution* are defined as usual. $T_{\Sigma \cup Var}$ denotes the set of terms built over Σ and Var. T_Σ is the set of ground terms (without variables) over Σ. For a term t, $Var(t)$ is the set of variables of t, $Pos(t)$ is the set of positions of t.

Example 1. Let Σ be a set of functional symbols such that $\Sigma = \{f, g, a, b\}$ where $ar(f) = 2$, $ar(g) = 1$ and $ar(a) = ar(b) = 0$. Let Var be a set of variables such that $Var = \{x, y\}$. Let t and t' be two terms defined such that $t = f(g(a), x)$ and $t' = f(g(a), b)$. Thus, $t \in T_{\Sigma \cup Var}$ and $t' \in T_\Sigma$. Moreover, $Var(t) = \{x\}$ and $Pos(t) = Pos(t') = \{\epsilon, 1, 11, 2\}$.

In the following, we consider the framework of *pure logic programming*, and the class of synchronized tree-tuple languages defined by CS-clauses [7,8].

Let us recall some usual definitions. We consider a set of predicate symbols $Pred$. $P^{\backslash n} \in Pred$ means that P is a predicate symbol and its arity is n. Given the set of predicate symbols $Pred$, $P^{\backslash n} \in Pred$ and $t_1, \ldots, t_n \in T_{\Sigma \cup Var}$, $P(t_1, \ldots, t_n)$ forms an atom. A Horn clause $H \leftarrow B$ is composed of a head H and a body B. The head of a clause is an atom and the body is a sequence of atoms that can be empty.

Example 2. Let $Pred$ be a set of predicate symbols such that $Pred = \{P_0^{\backslash 3}, P_1^{\backslash 1}, P_2^{\backslash 2}\}$. Let Σ be a set of functional symbols such that $\Sigma = \{f, g, a, b\}$ where $ar(f) = 2$, $ar(g) = 1$ and $ar(a) = ar(b) = 0$. Let Var be a set of variables such that $Var = \{x, y\}$. Thus,

- $P_0(f(x, g(y)), a, g(x)) \leftarrow P_1(x), P_2(b, y)$ is a Horn clause;
- $P_0(f(x, g(y)), a, g(x)) \leftarrow$ is also a Horn clause.

We will use letters G or B for sequences of atoms, and A or H for atoms. We denote by $A \in G$ that the atom A occurs in the sequence G. We also denote by $B \subseteq G$ that B is a sequence of atoms whose each atom is an atom of G. $G \setminus B$ represents a sequence of atoms such that each atom of $G \setminus B$ occurs in G but not in B. An empty sequence of atoms is denoted by $\emptyset$. Let G be a sequence of atoms. We denote by $G[A \leftarrow B]$ the substitution of the atom A occurring in G by the sequence of atoms B.

Any set of Horn clauses is considered as a logic program.

2.1 Resolution

Definition 1. *Given a logic program Prog and a sequence of atoms G, G derives into G' by a* resolution *step if there exist a clause*[1] $H \leftarrow B$ *in Prog and an atom $A \in G$ such that A and H are unifiable by the most general unifier σ (then $\sigma(A) = \sigma(H)$) and $G' = \sigma(G)[\sigma(A) \leftarrow \sigma(B)]$. It is written $G \leadsto_\sigma G'$.*

Sometimes, we will write $G \leadsto_{[H \leftarrow B, \sigma]} G'$ to indicate the clause used by the resolution step.

Example 3. Let $Prog = \{P(x_1, g(x_2)) \leftarrow Q(x_1, x_2).\}$ be a logic program, and consider $G = P(f(x), y)$. Thus, $P(f(x), y) \leadsto_{\sigma_1} Q(f(x), x_2)$ with $\sigma_1 = [x_1/f(x), y/g(x_2)]$.

We consider the transitive closure $\leadsto^+$ and the reflexive-transitive closure $\leadsto^*$ of $\leadsto$. $G_1 \leadsto^*_{Prog} G_2$ means that G_2 can be derived from G_1 using the clauses of *Prog*.

Let $Mod(Prog)$ denote the least Herbrand model of *Prog*. It is well known that resolution is complete.

Theorem 1. *Let A be a ground atom. $A \in Mod(Prog)$ iff $A \leadsto^*_{Prog} \emptyset$.*

We also define the notion of language.

Definition 2. *Let Prog be a set of Horn clauses and P a predicate symbol. Let $\mathcal{L}_{Prog}(P)$ be the language of ground term-tuples recognized by P, defined by $\mathcal{L}_{Prog}(P) = \{\langle t_1, \ldots, t_n\rangle \mid P(t_1, \ldots, t_n) \in Mod(Prog)\}$.*

Example 4. Let *Prog* be a set of Horn clauses such that $Prog = \{P_0(s(x), s(y)) \leftarrow P_0(x, y)\ P_0(0, 0) \leftarrow\}$. Thus, $\mathcal{L}_{Prog}(P_0) = \{\langle s^n(0), s^n(0)\rangle \mid n \geq 0\}$.

Note that $\mathcal{L}_{Prog}(P) = \{\langle\rangle\}$ means that P has no parameter and moreover $P \leadsto^*_{Prog} \emptyset$. Thus, $\{\langle\rangle\} \neq \emptyset$.

2.2 CS-Program

CS-programs[2] are a particular syntactical class of logic programs [7,8] recognizing synchronized tree-(tuple) languages.

This class of language is closed under union and intersection with a regular language (in quadratic time) and has decidable membership and emptiness problems (in linear time). Synchronized tree-(tuple) languages include regular languages. Let B be a sequence of atoms. B is *flat* if for each atom $P(t_1, \ldots, t_n)$ of B, all terms $t_1, \ldots, t_n$ are variables. B is *linear* if each variable occurring in B (possibly at sub-term position) occurs only once in B. Note that the empty sequence of atoms (denoted by $\emptyset$) is flat and linear.

[1] We assume that the clause and G are renamed apart in order to have distinct variables.

[2] In former papers, synchronized tree-tuple languages were defined thanks to sorts of grammars, called constraint systems. Thus "CS" stands for Constraint System.

Definition 3. *A* CS-clause *is a Horn clause $H \leftarrow B$ s.t. B is flat and linear. A* CS-program *$Prog$ is a logic program composed of CS-clauses. $Pred(Prog)$ denotes the set of predicate symbols of $Prog$.*

For a CS-program $Prog$ and a predicate symbol P, $\mathcal{L}_{Prog}(P)$ is called a *synchronized language*.

2.3 Transforming Any Logic Program into a CS-Program

In [7,8], the authors have proposed a technique allowing to transform any logic program into an equivalent CS-program. However, their technique is a semi-algorithm in the sense that it may not terminate. In this section, we succinctly recall their technique, which is composed of two inference rules called *Unfolding* and *Definition introduction*.

Definition 4. *A* definition *is a pair of the form $P(x_1, \ldots, x_n) \triangleq B$, where P is a predicate symbol, B is a sequence of atoms, and $\{x_1, \ldots, x_n\} \subseteq Var(B)$.*

Semantically speaking, $P(x_1, \ldots, x_n) \triangleq B$ is considered as an equivalence. However, $\triangleq$ is not considered as symmetric, and B is called the body of the definition.

In their approach, given a logic program $Prog$, the authors initially define a set of new definitions in such a way that they associate a flat atom $P'(x_1, \ldots, x_n)$, where P' is new predicate symbol not occurring in $Prog$, with a flat atom $P(x_1, \ldots, x_n)$ where $P^{\backslash n}$ is a predicate symbol occurring in $Prog$. This is an initial configuration of D_{new} in the following. In such a way, considering that $Prog'$ is the result of the transformation of $Prog$ by their technique, the authors specify that they expect: $\mathcal{L}_{Prog'}(P') = \mathcal{L}_{Prog}(P)$.

In [7,8], a state is tuple $\langle Prog, D_{new}, D_{done}, C_{new}, C_{out}\rangle$ where $Prog$ is a set of Horn clauses, D_{new} and D_{done} are two sets of definitions, C_{new} is a set of clauses and C_{out} is a set of CS-clauses. Let S and S' be two states. We write $S \Rightarrow^U S'$ if S' can be deduced from S using the *Unfolding* rule (see Definition 5). We also write $S \Rightarrow^I S'$ if S' can be deduced from S using the *Definition introduction* rule (see Definition 8). We also write $S \Rightarrow S'$ that $S \Rightarrow^I S'$ or $S \Rightarrow^U S'$. The reflexive and transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. An initial state is of the form $\langle Prog, D_{new}, \emptyset, \emptyset, \emptyset\rangle$. A final state is of the form $\langle Prog, \emptyset, D_{done}, \emptyset, C_{out}\rangle$. In the following, $\uplus$ denotes disjoint union.

In a final state, C_{out} is the final CS-program resulting from the transformation of $Prog$ according to the initial set of definitions D_{new}.

Definition 5 (Unfolding [8]). *Pick a definition in D_{new}, select one or more of its body atoms according to some selection rules, and unfold them with all matching clauses of $Prog$. Formally:*

$$\frac{\langle Prog, D_{new} \uplus \{L \triangleq R \uplus \{A_1, \ldots, A_n\}\}, D_{done}, C_{new}, C_{out}\rangle}{\langle Prog, D_{new}, D_{done} \cup \{L \triangleq R \uplus \{A_1, \ldots, A_n\}\}, C_{new} \cup C, C_{out}\rangle}$$

where C is the set of all clauses $\mu(L \leftarrow R \cup B_1 \cup \ldots \cup B_n)$ such that each $H_i \leftarrow B_i$ is a clause in $Prog$ for $i = 1, \ldots, n$ and such that the simultaneous most general unifier μ of $(A_1, \ldots, A_n)$ and $(H_1, \ldots, H_n)$ exists.

Example 5. $\langle\{Q(s(x),s(y)) \leftarrow Q(x,y)\},\{P(z) \triangleq Q(z,z)\},\emptyset,\emptyset,\emptyset\rangle$
$\Rightarrow^U \langle\{Q(s(x),s(y)) \leftarrow Q(x,y)\},\emptyset,\{P(z) \triangleq Q(z,z)\},\{P(s(y)) \leftarrow Q(y,y)\},\emptyset\rangle$.

Basically, the *Unfolding* rule generates new clauses. These new clauses are then handled by the *Definition introduction* rule in order to enrich the output program C_{out}.

The *Definition introduction* rule either introduces a new predicate symbol, or reuses a predicate symbol being the head of an existing definition.

Let us explain the aim of the *Definition introduction* rule. Its role is to transform a non-CS-clause of C_{new} into a CS-one, by replacing its body by a flat and linear body, using a new or an existing definition. Some preliminary definitions are needed.

Definition 6 ([8]). *Let $H \leftarrow B$ be a clause, and let B' be a subset of B such that $Var(B') \cap Var(B \setminus B') = \emptyset$ (i.e. B' shares no variables with the rest of the body). A definition $L \triangleq R$ matches B' if there exists a variable renaming ρ for R such that $\rho R = B'$ and $Var(B') \setminus Var(H) = Var(\rho R) \setminus Var(\rho L)$. And we say that the definition matches the body atoms via ρ.*

From a given sequence of atom B, we need to extract some sub-sequences of atoms sharing variables. More precisely, if an atom is in such a sub-sequence, either it is the only one within or it contains at least one variable such that there is another atom in the same sub-sequence sharing this variable. But, for any extracted sub-sequences B' and B'', one has $Var(B') \cap Var(B'') = \emptyset$. The following definition explains how we build such sub-sequences.

Definition 7. *We define $\leftrightsquigarrow$ as being the smallest equivalence relation between atoms such that for all atoms A, A', $Var(A) \cap Var(A') \neq \emptyset \implies A \leftrightsquigarrow A'$.*
An equivalence class under $\leftrightsquigarrow$ is called a chain.

For an atom set or a clause or a definition t, $Chains(t)$ denotes the set of chains contained in the atom set t or in the body of the clause or of the definition t.

Example 6. $Chains(P(x,y,v) \leftarrow P(x,z),Q(y),P(x,v),Q(z)) = \{\{P(x,z), Q(z), P(x,v)\},\{Q(y)\}\}$.

Definition 8 (Definition introduction [8])

$$\frac{\langle Prog, D_{new}, D_{done}, C_{new} \uplus \{H \leftarrow B_1 \uplus \ldots \uplus B_n\}, C_{out}\rangle}{\langle Prog, D_{new} \cup D, D_{done}, C_{new}, C_{out} \cup \{H \leftarrow L_1,\ldots,L_n\}\rangle}$$

such that $Chains(B_1 \uplus \ldots \uplus B_n) = \{B_1,\ldots,B_n\}$, and for each $i \in \{1,\ldots,n\}$

$$L_i = \begin{cases} \rho L & \text{if } D_{done} \text{ contains a definition } L \triangleq R \text{ matching } B_i \text{ via } \rho \\ P_i(x_1,\ldots,x_k) & \text{otherwise, where } P_i \text{ is a new predicate symbol and} \\ & \{x_1,\ldots,x_k\} = Var(B_i) \cap Var(H). \end{cases}$$

and D is the set of all new definitions $P_i(x_1,\ldots,x_k) \triangleq B_i$.

Example 7. Let us consider Example 5 again. Using Unfolding and Definition introduction rules, we get:

$\langle\{Q(s(x),s(y)) \leftarrow Q(x,y)\},\{P(z) \triangleq Q(z,z)\},\emptyset,\emptyset,\emptyset\rangle$
$\Rightarrow^U \langle\{Q(s(x),s(y)) \leftarrow Q(x,y)\},\emptyset,\{P(z) \triangleq Q(z,z)\},\{P(s(y)) \leftarrow Q(y,y)\},\emptyset\rangle$
$\Rightarrow^I \langle\{Q(s(x),s(y)) \leftarrow Q(x,y)\},\emptyset,\{P(z) \triangleq Q(z,z)\},\emptyset,\{P(s(y)) \leftarrow P(y)\}\rangle$. Note that $C_{out} = \{P(s(y)) \leftarrow P(y)\}$ is a CS-program.

A complete example can be found in [8], Example 10.

3 Making the *Limet-Salzer* Technique [8] Terminate

We propose a full example in order to show that this technique may not terminate even on simple inputs.

Example 8. We want to transform the following logic program *Prog* according to an initial set of definitions D_{new} defined such that $D_{new} = \{G \triangleq P_0\}$.

$$Prog = \left\{\begin{array}{ll} P(s(x),s(y),z) \leftarrow P(x,y,z); & P(x,y,z) \leftarrow P(s(x),s(y),z); \\ P(0,s(x),0) \leftarrow; & P(s(x),0,0) \leftarrow; \\ P_0 \leftarrow P(0,0,0) & \end{array}\right\}$$

An expert reader may note that *Prog* is not a CS-program and $\mathcal{L}_{Prog}(P_0) = \emptyset$. Consequently, if the transformation of *Prog* results in a CS-program $Prog'$ then $\mathcal{L}_{Prog'}(G)$ will be empty as well.

So, let us define D^0_{new} such that $D^0_{new} = \{G \triangleq P_0\}$. We start from the initial state $S^0 = \langle Prog,\{G \triangleq P_0\},\emptyset,\emptyset,\emptyset\rangle$. In the following, we use semi-column to separate clauses and definitions in their respective sets.

$\langle Prog,\{G \triangleq P_0\},\emptyset,\emptyset,\emptyset\rangle$
$\Rightarrow^U \langle Prog,\emptyset,D^0_{done},\{G \leftarrow P(0,0,0)\},\emptyset\rangle$
where $D^0_{done} = \{G \triangleq P_0\}$
$\Rightarrow^I \langle Prog,\{P_{new_0} \triangleq P(0,0,0)\},D^0_{done},\emptyset,\{G \leftarrow P_{new_0}\}\rangle$
$\Rightarrow^U \langle Prog,\emptyset,D^1_{done},\{P_{new_0} \leftarrow P(s(0),s(0),0)\},\{G \leftarrow P_{new_0}\}\rangle$
where $D^1_{done} = D^0_{done} \cup \{P_{new_0} \triangleq P(0,0,0)\}$
$\Rightarrow^I \langle Prog,\{P_{new_1} \triangleq P(s(0),s(0),0)\},D^1_{done},\emptyset,\{G \leftarrow P_{new_0};\ P_{new_0} \leftarrow P_{new_1}\}\rangle$
$\Rightarrow^U \langle Prog,\emptyset,D^2_{done},\{P_{new_1} \leftarrow P(s^2(0),s^2(0),0);\ P_{new_1} \leftarrow P(0,0,0)\},C^2_{out}\rangle$
where $D^2_{done} = D^1_{done} \cup \{P_{new_1} \triangleq P(s(0),s(0),0)\}$
and $C^2_{out} = \{G \leftarrow P_{new_0};\ P_{new_0} \leftarrow P_{new_1}\}$
$\Rightarrow^I \langle Prog,\{P_{new_2} \triangleq P(s^2(0),s^2(0),0)\},D^2_{done},\emptyset,C^3_{out}\rangle$
where $C^3_{out} = \{G \leftarrow P_{new_1};\ P_{new_1} \leftarrow P_{new_2}\}$
$\Rightarrow^* \ldots$

So the technique does not terminate because each new step $\Rightarrow^I$ introduces a new definition of the form $P_{new_i} \triangleq P(s^i(0),s^i(0),0)$.

Now, we show in Sects. 3.2 and 3.3 how one can make the technique proposed in [8] terminate using the generalization process introduced in Sect. 3.1.

3.1 Generalization

In Definition 9, we introduce the process of *Generalization* inspired from [9]. In a few words, generalization transforms a clause into another one, whose least Herbrand model includes the original one.

Definition 9. *A mapping from the set of clauses to itself is called* generalization strategy *(and denoted* Gen*), if for each clause $H \leftarrow B$, there exists a substitution σ s.t. $\sigma(\mathsf{Gen}(H \leftarrow B)) = (H \leftarrow B)$.*

For example, $P(f(x)) \leftarrow Q(g(x))$ can be generalized into $P(f(x)) \leftarrow Q(y)$.

Lemma 1 shows that for each clause that could be generalized, a resolution step involving it before transformation still able to be performed after generalization process.

Lemma 1. *Let Prog be a logic program, $H \leftarrow B$ be a clause, and G be a sequence of atoms. Let us write $Prog_1 = Prog \cup \{H \leftarrow B\}$ and $Prog_2 = Prog \cup \{\mathsf{Gen}(H \leftarrow B)\}$.*

If $G \leadsto^*_{Prog_1} \emptyset$, then $G \leadsto^*_{Prog_2} \emptyset$.

Proof. By induction on the length n of the derivation $G \leadsto^*_{Prog_1} \emptyset$.

- $n = 0$. Then $G = \emptyset$, therefore $G \leadsto^*_{Prog_2} \emptyset$.
- $n \geq 1$.

If $G \leadsto^*_{Prog_1} \emptyset$ does not use the clause $H \leftarrow B$, then $G \leadsto^*_{Prog} \emptyset$, consequently $G \leadsto^*_{Prog_2} \emptyset$.

Otherwise, the derivation writes $G \leadsto^*_{Prog} G_1 \leadsto_{H \leftarrow B} G_2 \leadsto^*_{Prog_1} \emptyset$. Note that the length of $G_2 \leadsto^*_{Prog_1} \emptyset$ is strictly less than n. By induction hypothesis, we get $G_2 \leadsto^*_{Prog_2} \emptyset$.

On the other hand, since $G_1 \leadsto_{H \leftarrow B} G_2$, we get $G_1 \leadsto_{\mathsf{Gen}(H \leftarrow B)} G'_2$ and there exists a substitution σ s.t. $\sigma(G'_2) = G_2$. Moreover, since $G_2 \leadsto^*_{Prog_2} \emptyset$, we get $G'_2 \leadsto^*_{Prog_2} \emptyset$. Finally, $G \leadsto^*_{Prog} G_1 \leadsto_{\mathsf{Gen}(H \leftarrow B)} G'_2 \leadsto^*_{Prog_2} \emptyset$, i.e. $G \leadsto^*_{Prog_2} \emptyset$.

Note that using the same notations than in Lemma 1, $G \leadsto^*_{Prog_2} \emptyset \implies G \leadsto^*_{Prog_1} \emptyset$ does not hold. Let us illustrate it by a short example.

Example 9. Let $Prog$ be a logic program such that $Prog = \{P_0(f(x), y) \leftarrow Q(g(x), y);\ Q(g(a), b) \leftarrow;\ Q(c, d) \leftarrow\}$. Let Gen be a generalization strategy defined such that $\mathsf{Gen}(P_0(f(x), y) \leftarrow Q(g(x), y)) = P_0(f(x), y) \leftarrow Q(z, y)$. Let us compare $Prog$ and $Prog'$ where $Prog'$ is the logic program obtained from $Prog$ by replacing the clause $P_0(f(x), y) \leftarrow Q(g(x), y)$ by $\mathsf{Gen}(P_0(f(x), y) \leftarrow Q(g(x), y))$. Note that $P_0(f(a), c) \leadsto^*_{Prog'} \emptyset$ when $P_0(f(a), c) \not\leadsto^*_{Prog} \emptyset$. However, note also that $P_0(f(a), b) \leadsto^*_{Prog} \emptyset$ and $P_0(f(a), b) \leadsto^*_{Prog'} \emptyset$ as well.

Corollary 1. $Mod(Prog \cup \{H \leftarrow B\}) \subseteq Mod(Prog \cup \{\mathsf{Gen}(H \leftarrow B)\})$.

In our context, we just expect that the least Herbrand model of the generated program contains (and does not necessarily preserve) the least Herbrand model of the initial logic program. So, we may replace a clause by a more general one.

In [9], the authors want to preserve the least Herbrand model in an exact way. For a clause of the form $H \leftarrow A_1, \ldots, A_n, A''_1, \ldots, A''_m$, they define an atom sequence $A'_1, \ldots, A'_n$ that generalizes $A_1, \ldots, A_n$, i.e. there exists a substitution σ such that $\sigma(A'_1, \ldots, A'_n) = A_1, \ldots, A_n$. Then they define a new clause $GenP(x_1, \ldots, x_k) \leftarrow A'_1, \ldots, A'_n$ where $Var(A'_1, \ldots, A'_n) = \{x_1, \ldots, x_k\}$, and replace the initial clause by: $H \leftarrow \sigma(GenP(x_1, \ldots, x_k)), A''_1, \ldots, A''_m$ Thus, the least Herbrand model is preserved, but this does not help the technique proposed in [8] to terminate.

3.2 Integration of Generalization in [8]

Let Gen be a generalization strategy. Let us introduce a new inference rule called *Generalization*. Given a state $S = \langle Prog, D_{new}, D_{done}, C_{new}, C_{out}\rangle$, the idea is to transform a clause of C_{new} into a more general one.

Definition 10 (Generalization)

$$\frac{\langle Prog, D_{new}, D_{done}, C_{new} \uplus \{H \leftarrow B\}, C_{out}\rangle}{\langle Prog, D_{new}, D_{done}, C_{new} \uplus \{\mathsf{Gen}(H \leftarrow B)\}, C_{out}\rangle}$$

$S \Rightarrow^G S'$ means that S' is computed from S using the generalization rule. Consequently, we extend the notation $\Rightarrow$ in such a way: $S \Rightarrow S'$ iff $S \Rightarrow^I S'$ or $S \Rightarrow^U S'$ or $S \Rightarrow^G S'$.

3.3 Results

In this section, we show in Theorem 2 that, using the generalization process, we are able to compute a CS-program whose least Herbrand model includes the one of the original set of Horn clauses. We also show in Theorem 3 that the generalization process helps in making the technique proposed in [8] always terminate. As usual, *Prog* and Σ are assumed to be finite sets.

To prove Theorem 2, we first need to formalize the semantics of an arbitrary state, which will be preserved when applying an inference rule.

Notations: Consider a logic program *Prog*, a set of clauses C, and a set of definitions D. Let $Body(C) = \{B \mid (H \leftarrow B) \in C\}$, $Head(D) = \{H \mid H \triangleq B \in D\}$, $\hat{D} = \{(H \leftarrow B) \mid H \triangleq B \in D\}$. Note that $\hat{D}$ is a set of clauses.

Given a state $S = \langle Prog, D_{new}, D_{done}, C_{new}, C_{out}\rangle$, let $\hat{S} = Prog \cup \hat{D}_{new} \cup C_{new} \cup C_{out}$. Note that $\hat{S}$ does not include D_{done}.

Definition 11. *The state* $S = \langle Prog, D_{new}, D_{done}, C_{new}, C_{out}\rangle$ *is said* coherent *if*

- $Pred(C_{out}) \cap Pred(Prog) = \emptyset$,
- $Pred(Body(D_{new})) \subseteq Pred(Prog)$ *and* $Pred(Body(C_{new})) \subseteq Pred(Prog)$,
- *and for all predicate symbol* P *occurring in* $Head(D_{done})$, $\mathcal{L}_{\hat{D}_{done} \cup Prog}(P) \subseteq \mathcal{L}_{\hat{S}}(P)$ *(i.e.* D_{done} *is redundant),*
- *and every predicate symbol occurring in* $Head(D_{new} \cup D_{done})$ *occurs only once in* $\hat{D}_{new} \cup \hat{D}_{done}$ *and never in* $Prog$ *nor in* $Body(C_{new})$.

Example 10. Let $Prog$ be the set of clauses as defined in Example 8. Let S be a state such that $S = \langle Prog, D_{new}, D_{done}, C_{new}, C_{out}\rangle$ where $D_{new} = \emptyset$, $D_{done} = \{G \triangleq P_0;\ P_{new_0} \triangleq P(0,0,0);\ P_{new_1} \triangleq P(s(0), s(0), 0)\}$, $C_{new} = \{P_{new_1} \leftarrow P(s^2(0), s^2(0), 0);\ P_{new_1} \leftarrow P(0,0,0)\}$ and $C_{out} = \{G \leftarrow P_{new_0};\ P_{new_0} \leftarrow P_{new_1}\}$. Thus, $Pred(C_{out}) = \{G;\ P_{new_0};\ P_{new_1}\}$ and $Pred(Prog) = \{P_0;\ P\}$. One has also:

- $Head(D_{done}) = \{G;\ P_{new_0};\ P_{new_1}\}$;
- $\hat{D}_{done} \cup Prog = \{G \leftarrow P_0;\ P_{new_0} \leftarrow P(0,0,0);\ P_{new_1} \leftarrow P(s(0),\ s(0), 0);\ P(s(x), s(y), z) \leftarrow P(x,y,z);\ P(x,y,z) \leftarrow P(s(x), s(y), z);\ P(0, s(x), 0) \leftarrow;\ P(s(x), 0, 0) \leftarrow;\ P_0 \leftarrow P(0,0,0)\}$;
- $\hat{S} = \{P(s(x), s(y), z) \leftarrow P(x,y,z);\ P(x,y,z) \leftarrow P(s(x), s(y), z);\ P(0, s(x), 0) \leftarrow;\ P(s(x), 0, 0) \leftarrow;\ P_0 \leftarrow P(0,0,0);\ P_{new_1} \leftarrow P(s^2(0), s^2(0), 0);\ P_{new_1} \leftarrow P(0,0,0);\ G \leftarrow P_{new_0};\ P_{new_0} \leftarrow P_{new_1}\}$;
- $Head(D_{new} \cup D_{done}) = \{G;\ P_{new_0};\ P_{new_1}\}$;
- $\hat{D}_{new} \cup \hat{D}_{done} = \{G \leftarrow P_0;\ P_{new_0} \leftarrow P(0,0,0);\ P_{new_1} \leftarrow P(s(0), s(0), 0)\}$;
- $Body(C_{new}) = \{P(s^2(0), s^2(0), 0);\ P(0,0,0)\}$.

Thus, one can check that

- $Pred(C_{out}) \cap Pred(Prog) = \emptyset$;
- $Pred(Body(D_{new})) \subseteq Pred(Prog)$ and $Pred(Body(C_{new})) \subseteq Pred(Prog)$;
- For any $P \in Head(D_{done})$, $\mathcal{L}_{\hat{D}_{done} \cup Prog}(P) = \emptyset$ and $\mathcal{L}_{\hat{S}}(P) = \emptyset$. Consequently, $\mathcal{L}_{\hat{D}_{done} \cup Prog}(P) \subseteq \mathcal{L}_{\hat{S}}(P)$;
- For any $P \in Head(D_{new} \cup D_{done}) = \{G;\ P_{new_0};\ P_{new_1}\}$, P occurs only once in $\hat{D}_{done} \cup \hat{D}_{new}$ and moreover, $P \notin Pred(Prog) \cup Pred(Body(C_{new}))$.

Consequently, S is coherent.

Lemma 2. *If* S *is coherent and* $S \Rightarrow S'$, *then* S' *is coherent.*

Proof. We proceed by case analysis.

- Let us study the application of the Unfolding rule. Suppose that $S' = \langle Prog, D'_{new}, D'_{done}, C'_{new}, C'_{out}\rangle$ and that $S \Rightarrow^U S'$. Suppose also that S is coherent.
  - By Definition 5, $C'_{out} = C_{out}$. Since S is coherent, $Pred(C'_{out}) \cap Pred(Prog) = \emptyset$.

- By Definition 5, $D'_{new} \subset D_{new}$. Since S is coherent, one has $Pred(Body(D_{new})) \subseteq Pred(Prog)$. Consequently, one has $Pred(Body(D'_{new})) \subseteq Pred(Prog)$. On the other hand, according to Definition 5, $C'_{new} = C_{new} \cup C$. Since S is coherent, $Pred(Body(C_{new})) \subseteq Pred(Prog)$. Let us prove that $Pred(Body(C)) \subseteq Pred(Prog)$. C is composed of clauses of the form $\mu(L \leftarrow R \cup B_1 \cup \ldots \cup B_n)$. R is a sequence of atoms from $Body(D_{new})$ and $Pred(Body(D_{new})) \subseteq Pred(Prog)$ since S is coherent. $B_1, \ldots, B_n$ are sequences of atoms that come from $Body(Prog)$. Consequently, $Pred(B_1 \cup \ldots \cup B_n) \subseteq Pred(Prog)$. So, $Pred(Body(C'_{new})) \subseteq Pred(Prog)^{(a)}$.
- By definition $\hat{S}' = Prog \cup \hat{D}'_{new} \cup C'_{new} \cup C'_{out}$. According to Definition 5, $\hat{D}'_{new} = \hat{D}_{new} \setminus \{L \leftarrow R \uplus \{A_1, \ldots, A_n\}\}$. C'_{new} is built from the unfolding of the clause $L \leftarrow R \uplus \{A_1, \ldots, A_n\}$ according to $Prog$. So, one can deduce that the substitution of a clause by a set of clauses resulting of an unfolding step on this clause preserves the Herbrand model. Consequently, for all predicate symbol P occurring in $Head(D'_{done})$, $\mathcal{L}_{\hat{D}'_{done} \cup Prog}(P) \subseteq \mathcal{L}_{\hat{S}'}(P)$.
- According to Definition 5, $D_{new} \cup D_{done} = D'_{new} \cup D'_{done}$ and then $Head(D'_{new} \cup D'_{done}) = Head(D_{new} \cup D_{done})$. Consequently, every predicate symbol P occurring in $Head(D'_{new} \cup D'_{done})$ occurs only once in $\hat{D}'_{new} \cup \hat{D}'_{done}$. Thus, if S is coherent then for all predicate symbol P occurring in $Head(D'_{new} \cup D'_{done})$, P does not occur in $Prog$. Finally, from $^{(a)}$ every predicate symbol occurring in $Body(C'_{new})$ is a predicate symbol of $Pred(Prog)$. Consequently, every predicate symbol occurring in $Body(C'_{new})$ cannot be a symbol of $Head(D'_{new} \cup D'_{done})$.

Thus, if $S \Rightarrow^U S'$ and S is coherent then so is S'.

– Let us now study the application of the Definition-introduction rule. Suppose that $S' = \langle Prog, D'_{new}, D'_{done}, C'_{new}, C'_{out}\rangle$ and that $S \Rightarrow^I S'$. Suppose also that S is coherent. According to Definition 8, $D_{done} = D'_{done}$, $C'_{out} = C_{out} \cup \{H \leftarrow L_1, \ldots, L_n\}$, $C'_{new} = C_{new} \setminus \{H \leftarrow B_1 \uplus \ldots \uplus B_n\}$ and $D'_{new} = D_{new} \cup D$.
  - Since S is coherent, one has $Pred(C_{out}) \cap Pred(Prog) = \emptyset$. Consequently, one has to check $Pred(\{H \leftarrow L_1, \ldots, L_n\}) \cap Pred(Prog) \stackrel{?}{=} \emptyset$. According to Definition 8, one can deduce that each L_i has either a new predicate symbol or a predicate symbol occurring in $Head(D_{done})$. So, if L_i has a new predicate symbol then this symbol does not occur in $Pred(Prog)$. Otherwise, L_i has a predicate symbol occurring in $Head(D_{done})$. Since S is coherent, from fourth item of Definition 11, every predicate symbol P occurring in $Head(D_{new} \cup D_{done})$ never occurs in $Prog$. Consequently one can deduce that $Pred(C'_{out}) \cap Pred(Prog) = \emptyset$.
  - According to Definition 8, $D'_{new} = D_{new} \cup D$. Since S is coherent, $Pred(Body(D_{new})) \subseteq Pred(Prog)$. Let us prove $Pred(Body(D)) \subseteq Pred(Prog)$. $Body(D)$ is composed of atoms that come from $Body(C_{new})$. Since S is coherent, $Pred(Body(C_{new})) \subseteq Pred(Prog)$. So, $Pred(Body(D'_{new})) \subseteq Pred(Prog)$. On the other hand, from Definition 8, $C'_{new} \subset C_{new}$ and since S is coherent, one has $Pred(Body(C_{new})) \subseteq Pred(Prog)$. Consequently, $Pred(Body(C'_{new})) \subseteq Pred(Prog)$.

- According to [9], one can deduce that the definition introduction transformation does not change the Herbrand model from $\hat{S}$ to $\hat{S}'$. Consequently, one can deduce that for all predicate symbol P occurring in $Head(D'_{done})$, $\mathcal{L}_{\hat{D'}_{done} \cup Prog}(P) \subseteq \mathcal{L}_{\hat{S}'}(P)$.
- Let us study $Head(D'_{new} \cup D'_{done})$. By construction, one can deduce that $Head(D'_{new} \cup D'_{done}) = Head(D_{new}) \cup Head(D) \cup Head(D_{done})$. Let us now focus on $\hat{D'}_{new} \cup \hat{D'}_{done}$. Once again, by construction, one can deduce that $\hat{D'}_{new} \cup \hat{D'}_{done} = \hat{D}_{new} \cup \hat{D} \cup \hat{D}_{done}$. According Definition 8, the definitions occurring in D are all of the form $P_i(x_1, \ldots, x_k) \triangleq B_i$ and P_i is a new predicate symbol. Moreover, B_i is composed only of predicate symbols occurring in $Pred(Prog)$. For any predicate symbol P occurring in $Head(D_{new}) \cup Head(D_{done})$, P does not occur in $\hat{D}$. Since S is coherent, P occurs only once in $\hat{D}_{new} \cup \hat{D}_{done}$. It is also true that for any predicate symbol P_i in $Head(D)$, P_i is a new predicate symbol. So, P_i cannot occur anywhere else than within its current definition. Consequently, for any predicate symbol P occurring in $D'_{new} \cup D'_{done}$, P occurs only once in $\hat{D}'_{new} \cup \hat{D}'_{done}$. Trivially, one can deduce that P does not occur neither in $Prog$ nor in $Body(C'_{new})$.

Thus, if $S \Rightarrow^I S'$ and S is coherent then so is S'.

– Suppose that S is coherent and $S \Rightarrow^G S'$. The Generalization rule only replaces a clause $H \leftarrow B$ of C_{new} by $\mathsf{Gen}(H \leftarrow B)$, which does not change the predicate symbols. So, one can deduce that the first, second and fourth items of Definition 11. Trivially, if Now, let S' be defined such that $S' = \langle Prog, D_{new}, D_{done}, C'_{new}, C_{out} \rangle$, and let P be a predicate symbol occurring in $Head(D_{done})$. Then $\mathcal{L}_{\hat{D}_{done} \cup Prog}(P) \subseteq \mathcal{L}_{\hat{S}}(P)$ because S is coherent, and $\mathcal{L}_{\hat{S}}(P) \subseteq \mathcal{L}_{\hat{S}'}(P)$ thanks to Corollary 1. Thus $\mathcal{L}_{\hat{D}_{done} \cup Prog}(P) \subseteq \mathcal{L}_{\hat{S}'}(P)$.

Lemma 3. *Let $S = \langle Prog, D_{new}, D_{done}, C_{new}, C_{out} \rangle$ be a coherent state. Suppose $S \Rightarrow S'$ and let us write $S' = \langle Prog, D'_{new}, D'_{done}, C'_{new}, C'_{out} \rangle$. Then every predicate symbol P occurring in $Head(D_{new} \cup D_{done})$ also occurs in $Head(D'_{new} \cup D'_{done})$, and $\mathcal{L}_{\hat{S}}(P) \subseteq \mathcal{L}_{\hat{S}'}(P)$.*

Proof. It comes from [9] for Unfolding and Definition-introduction rules, and from Corollary 1 for the Generalization rule.

Theorem 2 (Extension of [8]). *Consider a strategy of generalization* Gen *and an initial state $S^0 = \langle Prog, D^0_{new}, \emptyset, \emptyset, \emptyset \rangle$ s.t. each predicate symbol occurring in $Head(D^0_{new})$ occurs only once in $D^0_{new} \cup Prog$. If there exists a state S^n such that $S^0 \Rightarrow^* S^n$ and $S^n = \langle Prog, \emptyset, D^n_{done}, \emptyset, C^n_{out} \rangle$, then for any definition $P(x_1, \ldots, x_n) \triangleq B \in D^0_{new}$, $\mathcal{L}_{Prog \cup \hat{D}^0_{new}}(P) \subseteq \mathcal{L}_{C^n_{out}}(P)$.*

Proof. Note that S^0 is coherent. From Lemma 2, S^n is also coherent. From Lemma 3, $\mathcal{L}_{Prog \cup \hat{D}^0_{new}}(P) = \mathcal{L}_{S^0}(P) \subseteq \mathcal{L}_{S^n}(P)$. However, since S^n is coherent, $\mathcal{L}_{S^n}(P) = \mathcal{L}_{C^n_{out}}(P)$. Consequently $\mathcal{L}_{Prog \cup \hat{D}^0_{new}}(P) \subseteq \mathcal{L}_{C^n_{out}}(P)$.

Let us show how generalization helps for the termination of [8].

Theorem 3 (Termination). *For any program Prog and any coherent initial state* $S^0 = \langle Prog, D^0_{new}, \emptyset, \emptyset, \emptyset \rangle$*, there always exists a strategy of generalization* Gen *and a state* S *such that* $S^0 \Rightarrow^* S^n$ *and* $S^n = \langle Prog, \emptyset, D^n_{done}, \emptyset, C^n_{out} \rangle$*.*

Proof. Consider an arbitrary coherent state $S = \langle Prog, D_{new}, D_{done}, C_{new}, C_{out} \rangle$. Applying once the unfolding rule on S strictly decreases the size of D_{new}. Then applying many times this rule on S necessarily terminates. Applying once the definition-introduction rule on S strictly decreases the size of C_{new}. Then applying many times this rule on S necessarily terminates. Therefore an infinite derivation using those rules necessarily includes infinitely many steps with the unfolding rule (and also with the definition-introduction rule), which makes D_{done} bigger and bigger (the heads of definitions are pairwise different since the states are coherent). Thus an infinite derivation makes D_{done} infinite.

Let Gen be the generalization strategy such that for every clause $H \leftarrow A_1\ (t^1_1, \ldots, t^1_{n_1}), \ldots, A_m(t^m_1, \ldots, t^m_{n_m})$, we have $\mathsf{Gen}(H \leftarrow A_1(t^1_1, \ldots, t^1_{n_1}), \ldots, A_m(t^m_1, \ldots, t^m_{n_m})) = (H \leftarrow A_1(x^1_1, \ldots, x^1_{n_1}), \ldots, A_m(x^m_1, \ldots, x^m_{n_m}))$ where $x^1_1, \ldots, x^m_{n_m}$ are new fresh variables not occurring in $H \cup \bigcup_{i=1,\ldots,m} Var(A_i(t^i_1, \ldots, t^i_{n_i}))$.

Assume that a generalization step is applied as soon as a clause is added into C_{new}. Then every clause of C_{new} is of the form $H \leftarrow A_1(x^1_1, \ldots, x^1_{n_1}), \ldots, A_m(x^m_1, \ldots, x^m_{n_m})$, and note that $Chains(A_1(x^1_1, \ldots, x^1_{n_1}), \ldots, A_m(x^m_1, \ldots, x^m_{n_m})) = \{\{A_1(x^1_1, \ldots, x^1_{n_1})\}, \ldots, \{A_m(x^m_1, \ldots, x^m_{n_m})\}\}$.

Indeed, the freshness of variables ensures that each variable introduced by Gen occurs only once.

Consequently, in a derivation starting from S^0, every definition added in D_{new}, and then in D_{done}, is of the form $P^i_{new} \triangleq A_i(x^1_i, \ldots, x^i_{n_i})$ and $A_i \in Pred(Prog)$. So, one can deduce that using such a strategy, the size of D_{done} is bounded by $|D^0_{new}| + |Pred(Prog)|$. Therefore, the derivation issued from S^0 terminates and there exists a state $S^n = \langle Prog, \emptyset, D^n_{done}, \emptyset, C^n_{out} \rangle$ such that $S^0 \Rightarrow^* S^n$.

4 Example

In this section, we tackle a reachability problem described in [1], which cannot be handled successfully using regular approximations (using tree automata in [3] or tree transducers in [2]). This problem can be reduced to the problem $P_0 \overset{?}{\rightsquigarrow}_{Prog} \emptyset$ in Example 8. As we have shown in Example 8, the original technique does not terminate.

Now, consider the generalization strategy Gen defined by: for any clause $H \leftarrow P(s(0), s(0), 0)$, one has $\mathsf{Gen}(H \leftarrow P(s(0), s(0), 0)) = H \leftarrow P(x, x, z)$. We also define Gen for any clause of the form $H \leftarrow P(s(x), s(x), z)$ in such a

way: $\mathsf{Gen}(H \leftarrow P(s(x), s(x), z)) = H \leftarrow P(y, y, z)$. If we apply a generalization step after each unfolding step when it is possible, we get:

$$
\begin{array}{l}
\langle Prog, \{G \triangleq P_0\}, \emptyset, \emptyset, \emptyset\rangle \\
\Rightarrow^U \langle Prog, \emptyset, D^0_{done}, \{G \leftarrow P(0,0,0)\}, \emptyset\rangle \\
\text{where } D^0_{done} = \{G \triangleq P_0\} \\
\Rightarrow^G \langle Prog, \emptyset, D^0_{done}, \{G \leftarrow P(x,x,0)\}, \emptyset\rangle \\
\Rightarrow^I \langle Prog, \{P_{new_0} \triangleq P(x,x,0)\}, D^0_{done}, \emptyset, \{G \leftarrow P_{new_0}\}\rangle \\
\Rightarrow^U \langle Prog, \emptyset, D^1_{done}, \{P_{new_0} \leftarrow P(s(x), s(x), 0)\}, \{G \leftarrow P_{new_0}\}\rangle \\
\text{where } D^1_{done} = D^0_{done} \cup \{P_{new_0} \triangleq P(x,x,0)\} \\
\Rightarrow^G \langle Prog, \emptyset, D^1_{done}, \{P_{new_0} \leftarrow P(x,x,0)\}, \{G \leftarrow P_{new_0}\}\rangle \\
\Rightarrow^I \langle Prog, \emptyset, D^1_{done}, \emptyset, \{G \leftarrow P_{new_0};\ P_{new_0} \leftarrow P_{new_0}\}\rangle
\end{array}
$$

The derivation stops with $C_{out} = \{G \leftarrow P_{new_0};\ P_{new_0} \leftarrow P_{new_0}\}$. Note that $G \not\rightarrow_{C_{out}} \emptyset$ and consequently, $\mathcal{L}_{C_{out}}(G) = \emptyset$. So, thanks to Theorem 2, one can deduce that $\mathcal{L}_{Prog}(P_0) = \emptyset$.

We propose also a toy example but more practical in the sense that we want to check a safety property on a simple program.

Example 11. Suppose there is the following sequence of instructions in programming.

```
int x,y,xold,yold,z;
0. x <- readInt();
1. y <- readInt();
2. xold <- x
3. yold <- y
4. z <- x
5. x <- y
6. y <- z
```

The variables `xold` and `yold` store the initial values of `x` and `y`. We swap the values of `x` and `y` thanks to the variable `z`. Then we want to check that the last value of the variable `y` (resp. `x`) is actually the initial value of `x` (resp. `y`). It can be reformulated in the following way: there do not exist two integers such that, at the end of the execution, `xold != y` (resp. `yold != x`) i.e. the values haven't been switched since `y` (resp. `x`) should store the old value of `x` (resp. `y`).

We can specify the above program by a logic program $Prog$ composed of the following clauses:

1. $P_{readInt}(s(x)) \leftarrow P_{readInt}(x)$
2. $P_{readInt}(0) \leftarrow$
3. $P_{\neq}(s(x), s(y)) \leftarrow P_{\neq}(x, y)$
4. $P_{\neq}(s(x), O) \leftarrow$
5. $P_{\neq}(0, s(y)) \leftarrow$
6. $P_0(x_{old}, x, y_{old}, y, z) \leftarrow P_{readInt}(x)$
7. $P_1(x_{old}, x, y_{old}, y_{new}, z) \leftarrow P_0(x_{old}, x, y_{old}, y, z), P_{readInt}(y_{new})$
8. $P_2(x, x, y_{old}, y, z) \leftarrow P_1(x_{old}, x, y_{old}, y, z)$

9. $P_3(x_{old}, x, y, y, z) \leftarrow P_2(x_{old}, x, y_{old}, y, z)$
10. $P_4(x_{old}, x, y_{old}, y, x) \leftarrow P_3(x_{old}, x, y_{old}, y, z)$
11. $P_5(x_{old}, y, y_{old}, y, z) \leftarrow P_4(x_{old}, x, y_{old}, y, z)$
12. $P_6(x_{old}, x, y_{old}, z, z) \leftarrow P_5(x_{old}, x, y_{old}, y, z)$
13. $P_{check}(x_{old}, y) \leftarrow P_6(x_{old}, x, y_{old}, y, z), P_{\neq}(x_{old}, y)$

The clauses 1 and 2 specify the instruction `readInt()` in such a way that it allows the construction of any peano integer. The clauses 3, 4 and 5 specify the inequality test between two peano integers. Note that $P_{\neq}(x, x) \not\rightsquigarrow^* \emptyset$. The clauses 6 to 12 encode the execution of the sequence of instructions from the program point 0 to the program point 6. For $i = 0, \ldots, 6$, P_i specifies the program state by storing the values of the five variables at the program point i. Finally, Clause 13 specifies the property we want to check: do there exist two different integers specifying that, at the end of the execution, `y` does not store the old value of `x`.

Note that *Prog* is not a CS-program. Indeed, Clause 13 has a non linear body, consequently, it is not a CS-clause.

Consider the initial set of definitions $D_{new} = \{G(x, y) \triangleq P_{check}(x, y)\}$. We have developed a prototype requiring human interactions for unfolding or generalization. This prototype has been developed in Java and takes as input an XML file specifying: the program to transform, a set of new definitions (D_{new}) and a set of generalization rules. Note that for this example, the generalization is not needed for ensuring termination. But the motivation of this example is to show that the nature of CS-program can be useful for program verification.

The output program of our prototype is the following:

```
**********************************************
No more new definition to unfold !
The CS-program obtained is:
[G(x#43,x#46)<-Pnew0(x#43,x#46),
Pnew0(x#89,x#93)<-Pnew1(x#89,x#93),
Pnew1(x#135,x#139)<-Pnew2(x#135,x#139),
Pnew2(x#181,x#182)<-Pnew3(x#181,x#182),
Pnew3(x#227,x#228)<-Pnew4(x#227,x#228),
Pnew4(x#274,x#274)<-Pnew5(x#274),
Pnew5(x#319)<-Pnew6(x#319),Pnew7,
Pnew6(x#364)<-Pnew8(x#364),
Pnew7<-Pnew7, Pnew7<-,
Pnew8(s(x#453))<-Pnew9(x#453),
Pnew8(0)<-Pnew10,
Pnew9(x#501)<-Pnew8(x#501)
Pnew10 <- fail
]
```

The language of the obtained CS-program is empty. Consequently, the property is satisfied. Note that this example cannot be handled with regular approximations (tree automata or tree transducers for instance). Note that, in the output, we have obtain the clause $P_{new_{10}} \leftarrow fail$. After analysing the output trace of our prototype, we observe that in $P_{new_{10}} \triangleq P_{\neq}(0, 0) \in D_{done}$. Since

$P_{\neq}(0,0)$ cannot be unfolded with $Prog$, we associate the `fail` value meaning that the resolution does not succeed.

5 Discussion

The main difficulty is to define good generalization strategies, which ensure termination while computing a good approximation of the least Herbrand model. Even if some heuristics may generate some generalization rules, i.e. detecting equalities of sub-terms and then representing those sub-terms by a same variable, it remains difficult to control the computation without loosing the non-regularity of the language.

In [5], the authors have combined tree automata, abstract interpretation and Horn clauses. More precisely, they transform a set of Horn clauses into tree automata. Then, they apply some refinement techniques on those automata by eliminating some unfeasible traces detected after abstract interpretation of clauses. Those new automata produce a new set of clauses and so on... An interesting point in their approach is the way they use abstract interpretation. We focus on the quality of the approximation in terms of language, but we may have to reason in terms of quality of the analysis. Basically, Example 8 may be more easier to solve if we consider that the clauses $P(s(x), s(y), z) \leftarrow P(x, y, z)$ and $P(x, y, z) \leftarrow P(s(x), s(y), z)$ preserve the equality or inequality between the two first parameters. Then, starting from $P_0 \leftarrow P(0,0,0)$, abstract interpretation may help in order to show that clauses $P(0, s(x), 0) \leftarrow$ and $P(s(x), 0, 0) \leftarrow$ are never concerned by the unfolding process. Another point of view is that an analysis of the starting program may provide some clues in order to decide where the generalization, in our approach, can be used for preserving relations between terms.

6 Conclusion

CS-programs are particular logic programs that recognize synchronized tree languages (which include regular tree languages). This class of languages is closed under intersection with a regular language and emptiness is decidable. The technique described in [8] attempts to transform any logic program into an equivalent CS-program. This technique being a semi-algorithm, the computation may not terminate. We propose an extension using generalization strategies, to make the computation always terminate. However, the obtained CS-program may have a least Herbrand model larger than the initial one. In this way, we can deal with reachability problems and show that some terms are not reachable, as in Sect. 4, whereas it cannot be proved using the initial technique or with regular approximations [1].

However, the main difficulty is to define good generalization strategies, which ensure termination while computing a good approximation of the least Herbrand model. We intend to develop some heuristics to do it.

References

1. Boichut, Y., Héam, P.-C.: A theoretical limit for safety verification techniques with regular fix-point computations. Inf. Process. Lett. **108**(1), 1–2 (2008)
2. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular (tree) model checking. J. Softw. Tools Technol. Transf. **14**(2), 167–191 (2012)
3. Genet, T.: Decidable approximations of sets of descendants and sets of normal forms. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 151–165. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0052368
4. Gouranton, V., Réty, P., Seidl, H.: Synchronized tree languages revisited and new applications. In: Honsell, F., Miculan, M. (eds.) FoSSaCS 2001. LNCS, vol. 2030, pp. 214–229. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45315-6_14
5. Kafle, B., Gallagher, J.P.: Tree automata-based refinement with application to horn clause verification. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 209–226. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_12
6. Limet, S., Réty, P.: E-unification by means of tree tuple synchronized grammars. Discrete Math. Theor. Comput. Sci. **1**(1), 69–98 (1997)
7. Limet, S., Salzer, G.: Proving properties of term rewrite systems via logic programs. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 170–184. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25979-4_12
8. Limet, S., Salzer, G.: Tree tuple languages from the logic programming point of view. J. Autom. Reason. **37**(4), 323–349 (2006)
9. Pettorossi, A., Proietti, M.: Transformation of logic programs. In: Gabbay, D.M., Hogger, C.J., Robinson, J.A. (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, pp. 697–787. Oxford University Press, Oxford (1998)

Author Index

Ariola, Zena M. 205
Arrada, Imad-Seddik 1

Boichut, Yohan 245

Downen, Paul 205
Durán, Francisco 76, 98, 184

Eker, Steven 98
Escobar, Santiago 26, 98

Garavel, Hubert 1
Genet, Thomas 153
Gillard, Tristan 153

Haudebourg, Timothée 153

Johnson-Freyd, Philip 205

Lê Cong, Sébastien 153
Liu, Si 136

Martín, Óscar 59
Martí-Oliet, Narciso 59, 98
Meseguer, José 98, 136, 164, 184

Olarte, Carlos 115
Ölveczky, Peter Csaba 136

Pelletier, Vivien 245
Pimentel, Elaine 115

Ramírez, Sergio 226
Réty, Pierre 245
Riesco, Adrián 45
Rocha, Camilo 76, 115, 184, 226
Romero, Miguel 226

Salaün, Gwen 76

Tabikh, Mohammad-Ali 1
Talcott, Carolyn 98

Valencia, Frank 226
Verdejo, Alberto 59

Wang, Qi 136